Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2013

José Miguel Martins Lopes Mendes

Técnicas de segurança para a Internet das Coisas

Security techniques for the Internet of Things

Universidade de Aveiro
2013

Departamento de Eletrónica,
Telecomunicações e Informática

José Miguel Martins Lopes Mendes

Técnicas de segurança para a Internet das Coisas

Security techniques for the Internet of Things

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Rui Luís Andrade Aguiar, Professor associado c/ agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Paulo Jorge Salvador Serra Ferreira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor André Ventura da Cruz Marnoto Zúquete
professor auxiliar da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Paulo Alexandre Ferreira Simões
professor auxiliar da Faculdade de Ciências e Tecnologia da Universidade de Coimbra (arguente principal)

Prof. Doutor Rui Luís Andrade Aguiar
professor associado c/agregação da Universidade de Aveiro (orientador)

agradecimentos /
acknowledgements

Ninguém pode crescer e evoluir sem a interação e estímulo de diversas pessoas. No final desta etapa cabe-me atribuir o devido mérito e agradecer a todas as pessoas que contribuíram para me mudar e me tornar quem sou hoje e, consequentemente, ser capaz de escrever este trabalho.

Ao meu professor e orientador Rui Aguiar bem como ao professor João Paulo Barraca pela oportunidade que me deram de me juntar ao *Advanced Telecommunications and Networks Group* (ATNoG) onde pude aprender e expandir os meus horizontes de uma maneira que nunca teria conseguido sozinho nem noutro local.

A todos os membros do ATNoG pelas trocas de ideias, aprendizagem ou simplesmente conversas e perspetivas diferentes sobre a vida. Um especial obrigado ao Daniel Corujo pela colaboração, referências bibliográficas e conselhos.

A todos os meus professores ao longo deste caminho por todo o conhecimento transmitido.

A toda a minha família com um destaque especial para o meu pai que, apesar da distância que sempre nos separou, nunca deixou de estar presente e me apoiar.

Agradeço também a todos os meus colegas e amigos pelo apoio e amizade ao longo destes anos bem como paciência nesta última fase do percurso académico.

Palavras Chave IoT, M2M, Segurança, Criptografia, Sistemas Embutidos

Resumo

IoT assume que dispositivos limitados, tanto em capacidades computacionais como em energia disponível, façam parte da sua infraestrutura. Dispositivos esses que apresentam menos capacidades e mecanismos de defesa do que as máquinas de uso geral. É imperativo aplicar segurança nesses dispositivos e nas suas comunicações de maneira a prepará-los para as ameaças da Internet e alcançar uma verdadeira e segura Internet das Coisas, em concordância com as visões atuais para o futuro. Esta dissertação pretende ser um pequeno passo nesse sentido, apresentando alternativas para proteger as comunicações de dispositivos restritos numa perspetiva de performance assim como avaliar o desempenho e a ocupação de recursos por parte de primitivas criptográficas quando são aplicadas em dispositivos reais. Dado que a segurança em diversas ocasiões tem de se sujeitar aos recursos deixados após a implementação de funcionalidades, foi colocada uma implementação de exposição de funcionalidades, recorrendo ao uso de CoAP, num dispositivo fabricado com intenção de ser usado em IoT e avaliada de acordo com a sua ocupação de recursos.

Keywords IoT, M2M, Security, Cryptography, Embedded Systems

Abstract

IoT comprehends devices constrained in both computational capabilities and available energy to be a part of its infrastructure. Devices which also present less defense capabilities and mechanisms than general purpose machines. It's imperative to secure such devices and their communications in order to prepare them for the Internet menaces and achieve a true and secure Internet of Things compliant with today's future visions. This dissertation intends to be a small step towards such future by presenting alternatives to protect constrained device's communications in a performance related perspective as well as benchmarks and evaluation of resources used by cryptographic primitives when implemented on real devices. Due to security being on multiple occasions subjected to the resources available only after functionalities implementation, a minimalist implementation of functionalities exposure through the use of CoAP was also deployed in an IoT intended device and assessed according to resource overhead.

Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
ANACOM	Autoridade NAcional COMunicações (National Communications Authority)
AuC	Authentication Center
BTS	Base Transceiver Station
CBC	Cipher-Block Chaining
CFB	Cipher FeedBack
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CTR	CounTeR
DES	Data Encryption Standard
DoS	Denial of Service
DTLS	Datagram Transport Layer Security
ECB	Electronic CodeBook
ECC	Elliptic Curve Cryptography
EDGE	Enhanced Data rates for GSM Evolution
ETSI	European Telecommunications Standards Institute
GCC	GNU Compiler Collection
GEA	GPRS Encryption Algorithm
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile communications
HMAC	keyed-Hash Message Authentication Code
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IoT	Internet of Things
IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
IPSec	Internet Protocol Security
IV	Initialization Vector
LTE	Long Term Evolution
M2M	Machine to Machine
MAC	Message Authentication Code
MD5	Message Digest 5
MICKEY	Mutual Irregular Clocking KEYstream generator
MMU	Memory Management Unit
MSC	Mobile Switching Center
NIST	National Institute of Standards and Technology
OFB	Output FeedBack
OS	Operating System
RAM	Random-Access Memory
RC5	Rivest Cipher 5
REST	REpresentational State Transfer
RIPEMD	RACE Integrity Primitives Evaluation Message Digest
ROM	Read-Only Memory
SCL	Service Capability Layer
SD (card)	Secure Digital (card)
SHA	Secure Hash Algorithm
SIM	Subscriber Identity Module
SRAM	Static Random-Access Memory
TC	Technical Committee
TCB	Trusted Computing Base
TCP	Transmission Control Protocol
TEA	Tiny Encryption Algorithm

TLS	Transport Layer Security
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
VLR	Visitor Location Register
WEP	Wired Equivalent Privacy
WSN	Wireless Sensor Network
XTEA	eXtended TEA

Chapter One

Introduction

Since their invention, computer usage has spread from laboratories and research environments to slowly starting to infiltrate the common household. The advent of the Internet generated a whole new era of communications, public interest and information. Nowadays, computers take place in all our lives if, not for more, educating and entertaining us. Our connection to the Internet intensified even more with the advent of the Smartphone. Now computers don't limit themselves to be standing at our desks: they are always with us, 24/7 and we take them everywhere we go. The infiltration of computers could not have been more successful: they are in our homes, our workplaces, our pockets and what we do with our computers defines us and at a further extent, define our society and its people.

After so long and such large scale technology infiltration, it's finally time to fulfill the long awaited science fiction writing scenario: allow technology to interact with the real, physical world at a large scale. Even further, connecting that same technology to the Internet, allowing "smart" objects to interact with each other, exchanging information and collaborating. However, science fiction authors despite the obvious representation of futuristic scenarios, also seem to have a taste for tragedy and, disregarding the accuracy of the particular tragedies, it may be realistic to depict adversities and the upcoming of several problems. One thing is certain: the main future threats do not come from dystopian "killing machines" but instead from human beings itself.

Technology is not exactly known for having a clean security background when introducing new technologies. From communication protocols such as WEP and GSM to operating systems such as Android, it seems that first versions of new technologies are doomed to be successfully exploited. May it be for fun, profit or other reason, some people will try to tamper the new technological world offered by IoT, the sight of daily objects communicating and interacting through the Internet. Whereas in the computer world such behavior translates in attacks and malware limited to the virtual world, in IoT the possibility of inferring information and acting on reality is indeed intimidating (or it should be). Intimidating not only for the directly affected party (the end users) but also for companies manufacturing IoT-based solutions. Trust is a delicate property and, once lost, is very hard to regain. And while the

average user may find ordinary to have malware in his computer's hard drive, he may find containing security breaches (for example) in his home automation system or car that can be started from a Smartphone awfully shocking, resulting in a trust loss. The same can be applied to industrial and corporate IoT clients with the difference that there is a potential escalation in both direct and indirect damage. With IoT, technology has, once again, the opportunity to learn from past mistakes and to consider security with the same intensity that is applied towards functionalities avoiding future trust losses.

1.1 Objectives

First of all, this dissertation has as its main goal to provide practical guidance for anyone interested in implementing secure communications using restricted devices. Provide advice both in theoretical advantages/disadvantages and also practical results. Or at least be a good starting point for anyone concerned about diving further into one of the approached cryptographic options. To do so, the developed work aims to:

- Enhance the importance of security in wirelessly communicating restricted devices;
- Discuss advantages and disadvantages of using different cryptographic options as well as reinforcing the use of rational security— security adequate to security policies, scenarios and resources;
- Create a small, as platform agnostic as possible, cryptographic library capable of running in a panoply of devices with changes limited to a single point when porting to a different architecture:
  - Extraction of performance values from the created library when executing in real devices;
  - Selection of adequate algorithms for use in different IoT situations amongst the ones available.
- Evaluate the coexistence of functionalities and security, through the use of an application protocol placed in a restricted device and overhead measurement.

1.2 Contributions

The presented work was elaborated under the *Apollo* project, a collaboration between *PT Inovação* and *Instituto de Telecomunicações* which aims for the development of a platform capable of supporting M2M services. Allied to the possibility of developing generic applications on top of such services, the project also contemplates the implementation of two concrete and distinct scenarios: greenhouse monitoring and buses capable of detecting road holes.

This dissertation also originated an article entitled "Adaptative Security Measures for the Internet of Things" which resulted in a poster presented at StudECE 2013, submitting part of the results obtained using the developed cryptographic library (described in Chapter 4).

1.3 Structure

Chapter 2 starts by explaining the concept of IoT and end devices' properties, evolving then to a security related perspective where threats to these devices are identified. From there, an insight on the fragilities of restricted devices is presented as well as the scenario that IoT evolution must try to prevent at all costs: malware. After raising awareness on a conceivable alarming future and the urgency to secure devices and communications, the chapter concludes with the implications that secure messaging carries on the devices.

Reinforced the urgency of security in the previous chapter, **chapter 3** begins by explaining the importance of each device and device aggregates being secure as a whole. Afterwards, the compromise between cryptographic primitives and security policies is established, progressing to a discussion on cryptographic strategies, emphasizing advantages and disadvantages of different approaches when applied to restricted devices.

In **chapter 4** results from cryptographic algorithms when executed in real devices are presented. Firstly the methodology is described: the devices chosen and why, units of measurement, reference values as well as implementation specificities. After displaying the obtained results, the chapter finalizes with a discussion on the mentioned results.

Chapter 5 evaluates the cost of security when applied to a complete solution where devices do not contain the single task of performing cryptographic operations but also provide functionalities. An overview is presented on both the set of Technical Specifications produced by ETSI for M2M communications and CoAP. A CoAP implementation is then discussed as well as its overhead— which in some cases may reflect the amount of available resources for cryptographic primitives. An insight on the required capabilities for both security and functionalities is provided.

Finally, **chapter 6** concludes the dissertation with considerations about the developed work along with questions raised during its execution that can provide opportunities for further research.

Chapter Two

State of the art

At the present time, IoT is an extensively used term. News about products being manufactured, researched or announced is becoming increasingly common. Meanwhile, connecting one or a few embedded systems or smartphones to the Internet is not sufficient to achieve a true Internet of Things. Before tackling any other issues it's necessary, first of all, to define what exactly is this *Internet of Things*.

2.1 IoT is more than RFID

IoT is not a technology: it's a vision. A vision for the future itself where daily objects, *things*, can be connected and be a part of the Internet, a vision for the Future Internet. And perhaps the reason why there is not a globally accepted, accurate definition of IoT is because it's not common for different individuals to share the same exact vision about the upcoming future.

From the simple phrase "objects connected and interacting with the Internet", however, it's possible to elaborate a logical reasoning to extract some properties. If objects are connected to the Internet, a digital world, it's assumed they have digital capabilities. Ability to interact digitally also assumes processing capabilities. Objects can therefore be considered devices with processing and communication capabilities. The faculty of communication also assumes there is some sort of technology enabling communications: one or more technologies involved in connecting such devices to the Internet world. In fact the technologies enabling devices to connect to the Internet are not clear. Zorzi, Gluhak, Lange, *et al.* [1] compare IoT to the "wild west", an unexplored territory where all current technologies can play a role.

In the present time there are already (and have existed for a few years) technological concepts associated with the terms "objects communicating" and "machines interacting":

M2M Communication and technologies allowing interaction between machines (machine-to-machine).

RFID Radio-frequency identification is the use of radio communication as a mean to identify/track objects, persons or animals implanted with RFID tags— small wireless devices.

WSN Wireless sensor networks are networks consisting of autonomous nodes (devices) cooperating to gather/deliver information about the surrounding environment. Despite the word "sensor" in the name, nodes can also be actuators, interacting with the physical world based in human originated commands or other decision processes.

M2M is inherent to IoT as it is to WSNs: provides technology which enables direct machine interaction. IoT can also be viewed as an expansion or evolution of the concept of WSN: devices and aggregates of devices not only communicating and interacting inside an intranet but along the much more complex Internet. RFID has also been commonly associated with both WSNs and IoT due to the need for identification— it's necessary to identify objects/devices somehow— but IoT far extends the concept of simply identified devices by introducing a vision of globally connected objects.

With a vision comes also a paradigm. In fact, the IoT paradigm is a conjunction of other 3 paradigms/visions as identified by Atzori, Iera, and Morabito[2]:

- Things oriented— focus on a world where things communicate with computers and each other providing services
- Semantic oriented— focus on representing and exchanging gathered information, obtaining knowledge
- Internet oriented— focus on the global infrastructure to connect generic objects and the evolving resultant Internet

On top of such paradigms lay applications. Applications taking advantage of provided data, knowledge and communication being necessary a full combination of the referred three visions to unleash the true potential of applications relying on IoT[3].

2.2 IoT scenarios

Despite futuristic scenarios incited by the idea of objects communicating, there are present time areas of application where objects communicating and sharing information can be quite convenient such as:

Healthcare Measuring patients' vital signs over long periods of time and real-time monitoring, allowing patients to resume their daily lives while being diagnosed;

Tracking Tracking can be very useful and applied to multiple situations:

- Warehouses/Distribution centers— Companies can track their packages and find their products faster in warehouses if all are correctly identified;
- Lost keys— The scenario of not knowing the location of personal keys is quite common to some people. Keys transmitting their location can help mitigating the inconvenience of such situations;

- Pet tracking— Currently, animal RFID tags identify them allowing it's association to the rightful owners. Meanwhile there's still room to improvement as locating a troublesome runaway pet might be hard and real-time location could be quite helpful;
- People tracking— Despite the controversy, the fact is that brain degenerative disease patients such as Alzheimer patients can get lost and disoriented. Finding the location of lost relatives suffering from such disease may be imperative and devices providing their real-time location beneficial. Other uses of legitimate people tracking include children tracking, troublesome teenagers tracking and even paroled prisoners tracking.

City quality improvement IoT can help increasing the quality of life in urban centers by assessing:

- Road monitoring— Vehicles equipped with sensors can report road holes and imperfections. Reporting of such data in real-time or periodic intervals helps maintaining roads in adequate conditions;
- Pollution levels— Monitoring pollution levels in different city areas in order to investigate areas of action and increase population health is crucial in large urban centers ravaged by smog;
- Waste management— Currently, garbage containers are disposed at a given frequency (usually once per day at a specific time), resulting in containers being handled despite their load. Smart containers, could help reducing unnecessary trips (and costs) by informing waste management companies of their load. Informing about container load can also help companies managing the frequency of their disposals at specific areas.

Smart agriculture [1] Efficient agriculture, reducing production costs by monitoring parameters such as soil humidity, temperature and light conditions, respective remote monitoring and watering according to climate conditions;

Smart grids Intelligent distribution of energy according to consumer needs, creating a more sustainable and economic distribution. It may also help increasing the reliability of the service by balancing energy between clients being both beneficial to suppliers and clients;

Other energy saving scenarios There are many additional scenarios, besides smart grids, where IoT can help reducing energy costs:

- Smart lighting— Light is adjusted to the presence/absence of nearby humans and environment lighting;

[1] The word *smart* is inherent to multiple IoT scenarios, specially where decisions are based on environmental or past knowledge. The process of automated decisions is the reason for these scenarios being classified with such word.

- Smart air conditioning— Adjust temperature in rooms where humans are present or are likely to move to instead of heating the entire space;
- Smart power-off of devices— As much as people forget, standby devices consume energy. Televisions, routers, extension sockets and essentially every domestic equipment featuring a small light to indicate its activity consumes energy. In multiple cases there is no need for such devices to be in standby mode when the owner goes to sleep. Powering off such devices automatically during sleep periods may be beneficial for energy saving.

The myriad of other possible scenarios including but not limited to accurate weather prediction, larger distributed sensors networks for disaster prediction, road traffic monitoring, driver warnings and respective redirection and many more, provide IoT the potential of infiltrating itself in the quotidian human life bringing a digital revolution in connecting the physical and virtual worlds.

Meanwhile it should be noted that each scenario is unique in requirements whether they are reliability, security or power constrains. In healthcare and people tracking scenarios, security and reliability are priorities. Meanwhile, devices intending to save energy cannot consume more energy that the amount they save in order to fulfill their objective. At the same time, disaster prediction and real-time decisions have real-time constrains. In contrast, scenarios such as smart agriculture may or may not pose such tight constrains regarding real-time (environmental measures do not need to reach their destination immediately), reliability (there may exist periods where communication is not possible) and security (if an attacker intercepts sensor data, it will only collect information that he can obtain using its own sensors).

In fact, such heterogeneity of scenarios incites the pursue of a long-term architecture development. IoT scenarios and applications are only truly limited by a mix of imagination, usefulness and business opportunities, requiring an infrastructure prepared to deal with different scenario requirements and paradigm shift.

2.3 IoT horizontal architecture

In order to develop sustainable, long term IoT solutions there is the need and pursue for both horizontal architecture and standardization. Such is the vision of ETSI and other international standardization organisms resulting in a horizontal architecture where devices are at the bottom and services are exposed through a top layer. Applications can then be built using the services provided by the M2M infrastructure. Such horizontal approach is depicted in figure 2.1.

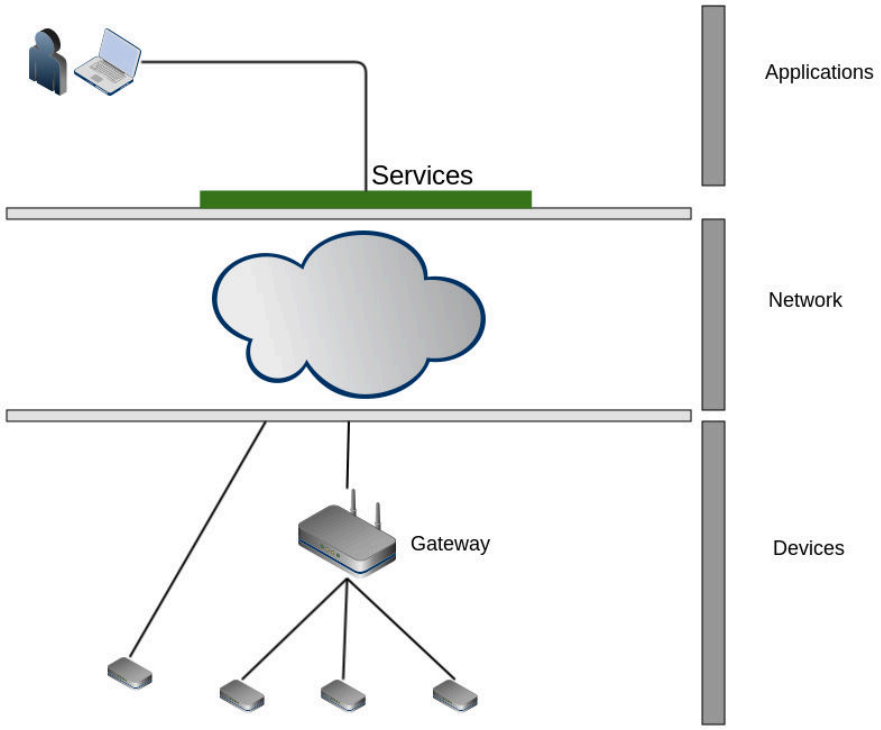


Figure 2.1: Simplified representation of the IoT horizontal approach.

2.4 IoT devices' characteristics

At the lower end of the IoT architecture lay devices— the actuators/sensors intended to gather information from or apply actions to the physical world. There is no consensus in what exactly defines an IoT device. Being IoT a vision instead of a technology or a set of specific technologies and, therefore, being classified by a mix of sociological and technological factors, a device may be classified accordingly to its intended use or through the specific role that it takes in a specific time interval. Smartphones and computers are, without a doubt, *things* and, in addition, they are considered daily objects. Meanwhile they are already a part of the current Internet not presenting a role in the IoT vision. If, however, they were to be equipped with sensors/actuators and the capability of acting on the real world they could be considered IoT devices. In fact, smartphones are already supplied with a variety of sensors, inferring information about the physical world. While a smartphone can infer information on the external world, it was not build with an IoT purpose in mind. There is a clear distinction from a smartphone that is showing its sensor's data to an user through its screen and a smartphone that is collaboratively working with other entities to provide a set of services available outside its own scope. It can be affirmed that a smartphone can or cannot be considered an IoT device considering the role it takes at specific time intervals and the same applies to all machines classified as general purpose machines (they can be considered IoT devices if they take such role).

IoT also comprehends the possibility of devices not built with a general purpose in mind

to be a part of its infrastructure. Due to the absence of general purpose characteristics, such devices can be classified as embedded systems. And, as embedded systems that they are, they may lack the same raw processing capabilities[2] and flexibility [4] as general purpose systems. Devices may be resource constrained in both:

- CPU— May be equipped with 8 or 16 bit microprocessors;
- RAM— It may not exceed the embedded RAM in the microprocessor;
- Permanent storage— as with RAM, may also be integrated into the microprocessor;
- Power— possibly running using batteries, without a continuous power source;
- Communications— Wireless communications only due to the impossibility of deploying cables scattered around the world. Radio communication may also be limited to a certain amount of maximum power, further narrowing the number of possible wireless protocols and range.

Such set of characteristics introduces further properties on the devices. Due to the low resources available, operating systems may or may not be available. Also, in addition to being possible or not, operating systems use both memory resources and CPU cycles— time which the CPU could spend being in idle mode and saving energy— bringing the question if it is or not desirable.

The lower boundary for devices can then be established at devices contain an 8 bit microprocessor with RAM and Flash/ROM integrated into the microprocessor, running on batteries, requiring periods of idle mode, without an operating system and including only a low-power radio interface. It's important to notice that the existence of a lower boundary does not, by all means, forbid the existence of devices with a much higher set of specifications and even the use of general purpose machines as IoT devices, as mentioned previously. However, in order to avoid constant repetitions of this fact, in the future (except when explicitly mentioned), devices will be considered embedded and constrained devices— the "worst case" scenario.

Still at the device level, there is another class of devices which are supposed to have a higher set of specifications than regular devices: gateways. Gateways are devices responsible for interlinking devices (which are not capable by themselves) with traditional communication networks[5]. As a bridge between what are basically WSNs and the Internet, the gateway is responsible to address:

- The heterogeneity of devices and respective different communication mechanisms;

[2]Embedded devices may contain ASICs (Application-Specific Integrated Circuits), integrated circuits designed for a specific use instead of general purpose. Such circuits may be extremely fast at performing the specialized task for which they were designed but lack the ability to run general tasks. Examples of such circuits would be circuits intended for sound or image processing.

- Data transmission and forwarding;
- Protocol conversion;
- Management and control of nodes (processing commands received from remote servers).

As such, due to the additional complexity as well as the need to communicate and process data from, possibly, a large number of nodes, gateway specifications as well as communication capabilities are set higher than devices specifications. In fact, it's not uncommon to use single board computers as the central units for gateways[5][6][7], setting a typical gateway as a device containing processing capabilities equivalent of multiple devices, with an operating system and a diversity of communication capabilities.

2.5 Security problems and countermeasures

Security problems and attacks are nothing new in the technological world as excessively boasted bulletproof systems have prove countless times that security is nothing to brag about as being impossible to bypass or break. IoT and WSNs, however, are still very far away to be secure enough to the point of inciting bragging due to a number of characteristics:

- Communications can be performed wirelessly: anyone can listen to them and communicate using the same medium;
- Physical access to the devices may be possible: devices can be placed in public places;
- Features devices with limited resources: range of security measures may be restricted.

Several possible attacks in WSNs and transposable to IoT have already being enumerated and described in the past. Kalita and Kar[8] identified no less than 37 possible attacks on such scenarios including but not limited to:

- Denial of service (DoS) attacks;
- Devices taking multiple identities (Sybil attack);
- Wormholes— tunneling of messages through a low latency link and replaying them in a different network location disrupting routing;
- Sinkhole— make a compromised node look attractive from a routing perspective, luring traffic to particular areas;
- Impersonation— attacker replicates the identification of a legitimate device;
- Eavesdropping— simply listening on the shared medium to discover communication contents;

- Traffic analysis— traffic patterns analysis can identify node roles and reveal activity of nodes[9];
- Attacks carried out by compromised devices inside the network;
- Invasive attacks (physical tampering) with the intent of reverse engineering, extracting keys or modifying code;
- Attacks performed by external devices with more capabilities than the nodes in the network.

There are not solutions for all these attacks. DoS attacks can only be minimized or prevented in specific situations such as, for example, not processing messages originating in devices who send a number of messages greater than a defined threshold (eventually using node resources). However, a strong willed attacker can always invest money in devices with more resources and therefore ability to create more collisions or simply devices dedicated to jam the entire wireless medium. This is an inherent problem of wireless communications which cannot be avoided: only measures to discourage weak-minded attackers can be taken.

One important measure to consider is the prevention of physical tampering. Physical tampering can go beyond the destruction/vandalization of the nodes to actually allow reverse engineering, obtainment of cryptographic keys and node reprogramming. Non-existence of physical protection may allow cryptographically trusted devices to perform malicious actions raising the question if cryptography should be applied when the secret keys can be obtained and there may be no point in discussing highly advanced and secure protocols which impact device resources if those same measures can be bypassed. The best analogy for this situation is the use of top-notch security locks in doors when a backup key is hidden under the doorway rug or similar. In scenarios where security importance overcomes the device importance, even destructive measures to the devices when attempted tampering should be considered. Except when explicitly stated, all future references to securing devices will assume that physical protection is well placed.

Use of secure messaging (messages secured using cryptographic measures), including at the routing layer, also reduce both data privacy questions and the number of successful attacks:

- Eavesdropping— if encryption is performed, attackers cannot discover the message contents without the decryption key.
- Impersonation— correctly authenticated messages cannot be forged.
- Sybil attack— as in impersonation, foreign devices cannot forge identities when authentication is required.
- Sinkhole— without being a part of the network (authenticated), attackers cannot send forged messages and therefore cannot lure traffic.

However, despite the use of cryptography, careful protocol design also plays a significant role (in fact a conclusion expressed by Kalita and Kar). A protocol intending to prevent attacks besides assuring communication, should also, for instance, only allow trusted parties to participate in routing decisions. A combination of cryptography and secure protocol design may also minimize traffic analysis where cryptography assures that the contents are not discovered and the protocol willingly tampers flow analysis such as assuring that all messages have the same size (tampering inference on the content) and the sending of bogus messages between devices (interfering with context inferences). In reactive systems, however, it may not be so simple as sudden changes in the surrounding environment may reveal a pattern of messages allowing inference on the message's purpose.

Wormhole attacks are very difficult to defend against since despite cryptography, replayed messages will still be valid. Higher level approaches— behavior approaches— can help detecting such attacks by detecting replicated nodes in different network locations. Routing protocols should be designed carefully in a way that vanquishes this attack scenario.

Despite the means of perpetration to compromise and subvert a node— physical tampering or one of other ways described further ahead— this situation should also be considered as possible. Detecting such situation is no trivial matter as the device can still function perfectly from the network's point of view but being compromised waiting for a trigger condition or communicating with a third-party outside the network's knowledge. One alarming scenario is the successfulness in compromising nodes responsible for alarms where the malfunctioning is discovered only after the alarm not fulfilling its role (being the alarm the trigger for non-intended/malicious behavior).

2.6 IoT is fragile

Even without considering physical tampering and wireless communications, restricted devices with limited resources and therefore lacking security features may be more vulnerable than regular general purpose machines. Networks of such nodes, designed to operate in an autonomous way may have trouble detecting compromised nodes or attacks. Exposing such devices and networks to the Internet world— where computers with multiple security features are successfully attacked and compromised— without a security plan and without considering the consequences can have catastrophic results.

2.6.1 Restricted devices are vulnerable

Memory protection.

25 years have passed since the widespread of the Morris Internet worm [3], the first computer worm that gained significant media attention. The fact is, however, that one of the

[3]Morris worm appeared in 1988 created by a student at MIT. Exploiting the possibility of buffer overflow in the *fingerd* daemon, it infected at least one twentieth of the existing computers at the time. Besides generating controversy, it also showed how insecure the Internet really was (and is) and the importance of security.[10]

security holes that allowed the Morris worm to be successful is still a reality today as it was 25 years ago: buffer overflows. After more than 2 decades of research and computational advances, classic buffer overflows are still ranked third on the CWE/SANS TOP 25 Most Dangerous Software Errors [11]. If memory errors have been a reality for so long it would be wise to assume that they will be present in the future. Memory errors exist, are currently being exploited for malicious purposes and will continue to be exploited as long as they exist [12]. To relinquish or minimize the threat that buffer overflows present, several techniques are present in nowadays computers:

ASLR ASLR randomizes locations in memory preventing injected code of successfully jumping to known addresses. Locations include libraries, stack, heap and the process itself.

Canaries Consists of placing random values at the beginning of a function and asserting that the value is still the same at the end of the function with the purpose of detecting stack-smashing. It can either be implemented by the compiler or manually during programming.

NX bit (Never eXecute) A technology implemented in some CPU families that allows portions of memory to be marked as non-executable. Data can safely be stored in these sections with the guarantee that will never be executed, rendering buffer overflows (with the purpose of code injection) on structures located in these regions meaningless.

Other MMU based techniques By dividing the memory in segments and accessing logical addresses instead of real memory addresses, an address translation is performed by a hardware unit known as MMU. Besides address translation, permissions can be defined over segments, not allowing one process without permissions to address segments not reserved for it.

From the techniques presented, almost every one relies on the existence of hardware capabilities and/or a residing operating system. Considering IoT devices, only one of the presented techniques could be applied in all the devices: canaries. Even so, depending on the compiler (if it introduces canaries or not), such capability may require additional programming effort. Restricted devices without hardware units dedicated to memory protection and without an operating system have very little defenses against buffer overflows.

CPU execution levels.

Nowadays in general purpose machines, CPUs contain multiple execution levels. These privilege levels, implemented in hardware, determine a set of possible actions that can be performed at each level and below. Most modern operating systems take advantage of such execution levels, clearly distinguishing the actions that regular users and the system can perform, forcing programs to invoke system components in order to perform privileged actions. This approach has the advantage of moderating access to resources, allowing access

to them only in a restricted, predefined way in which direct access to the resources is made with system code and not user code. The implementation of such model is performed with system calls. When an interruption is generated, it is handled by the operating system code (interruption handlers) which run at a privileged level, restoring the execution level back to normal on returning.

This, however, may not apply to IoT. Devices without an operating system may not have a set of predefined interruption handlers to moderate the access to resources. Even if they did, the enforcement of different execution levels in hardware would still be required. Otherwise, direct access to resources and even replacement of the interruption handlers in memory would still be possible. Different execution levels is a feature not available in many CPUs designed with embedded purpose in mind, effectively invalidating the resource access moderation meaning that successful injected code has access to all the device's resources.

2.6.2 Code injection

From the possible security problems that may affect IoT devices, one is particularly worrisome: code injection. Code injection goes beyond the point of disrupting the device's functioning (although it can also achieve this) to actually alter the device functioning in benefit of the attacker— the device will dedicate processing time and resources in order to fulfill the attacker's purpose.

Not considering physical access, there are at least 3 methods from which an attacker can induce code injection.

Buffer overflow. Impersonating a trusted entity or simply sending messages to the device an attacker can feed more data than the device is expecting, corrupting the memory with foreign code if a vulnerability is present. A very basic example of such vulnerability is present below:

```
char     buffer [100];
uint8_t pos = 0;

do{
    buffer[pos] = readChar();
}while(buffer[pos++] != '\0')
```

Without verifying array boundaries, a possible exploit that allows code injection may be introduced. By simply checking if the number of bytes read already reached the size of the array, the vulnerability would be removed from the example.

Faulty update mechanisms. Systems may consider the need or at least provide to future use, techniques that allow remote programming such as OTA [4]. If the techniques or imple-

[4]OTA or Over-the-air programming refers to techniques where reprogramming can be performed in end devices through wireless communication.

mentations are defected in a way that allows impersonation of a trusted entity responsible for updates, this can result in device programming by a third party and, therefore, code injection.

Compromised gateway. A gateway that has the ability of updating devices, controlled to a certain extent by an attacker can be used to introduce foreign code in the devices. This differs from the previous point since no impersonation is required and it doesn't require that vulnerabilities in the update protocol exist.

2.6.3 IoT: the next Android?

IoT growth predictions are astonishing regarding both future Internet traffic generated and number of manufactured devices. This situation is indeed akin to another: the booming of smartphones. Due to consumer interest, smartphones market penetration have been humongous and recent news can be found estimating the percentage of smartphone ownership around 60% in the United States[5].

Sadly, interest in mobile platforms did not only arouse from the average consumer but also from another unwanted audience: malware developers. Recent threat reports released by McAfee in the fourth quarter of 2012 and in the first quarter of 2013 show alarming numbers: besides an almost exponential spurt in mobile malware in the latest years [13], more than 1400 new unique malware families and variants for Android appeared in just one quarter[14], being this the most affected platform and revealing a tendency for an even greater increase in threats. Figures 2.2 and 2.3 represent the increase of mobile malware as well as the target platforms.

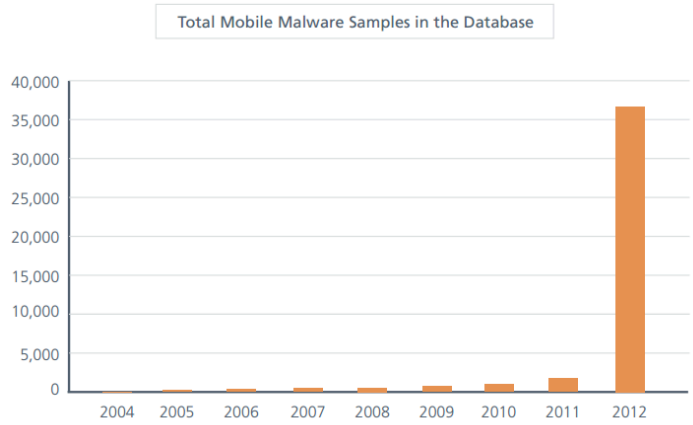


Figure 2.2: Increase in mobile malware in the latest years. [13]

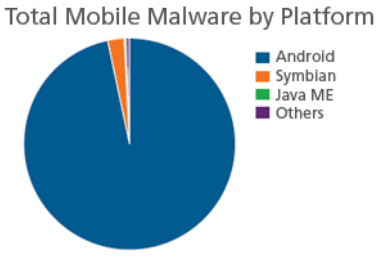


Figure 2.3: Distribution of mobile malware in the first quarter of 2013. [14]

[5]Number obtained through news released by the Nielsen company: `http://www.nielsen.com/us/en/newswire/2013/mobile-majority--u-s--smartphone-ownership-tops-60-.html` (last visited on 27-10-2013)

From the moment that a platform reaches mass consumption it becomes a desirable target for malware due to the large number of potential victims. In addition, mobile platforms present a unique set of sensitive information (contacts, GPS, passwords,...) making them an enticing target. Examining the current malware trend and the amount of data being threatened is enough to incite fear. And so, a major question arises: *Why not IoT?* If a device holding contacts and personal information (smartphone) raises so much interest from malware developers, a device that can be used, for instance, for home automation can also generate a significant amount of interest.

There is a very high probability that IoT devices will have vulnerabilities. It is also safe to assume that some people will find those same vulnerabilities and exploit them. If (or when) that happens, there will occur the need to counterattack and eliminate those threats.

2.6.4 Traditional malware detection applied to IoT

Prevention is always the best practice to embrace when designing secure systems. Nonetheless, the vision of prevention naturally also comprehends the possibility of the prevention itself failing arising the need to have a second strategy in place. In this context, this means that besides all safe programming rules and caution, systems can have security breaches leading to the inevitable question of which malware detection (and afterwards cleanse) techniques could be applied to IoT devices.

Regarding restricted devices without an operating system, the answer is very clear: the traditional approach of running tools for malware analysis does not work. Remembering that a single piece of code is executed (firmware), the code that would perform malware analysis would be included in that very same piece of code. However if the device were to be infected, it also would mean that the infected firmware would perform malware detection. This paradox may not be a very satisfying situation due to the level of trust and effectiveness that could be attributed to such detection.

More complex devices with more resources such as single board computers, however, allow this approach due to the ability of executing multiple processes, meaning that the code to analyze and the analyzer would be separated. Meanwhile there is the question of which malware detection technique would prove to be more efficient.

Static analysis.

Static malware analysis refers to techniques that infer if a given piece of code is malicious or not without actually executing it. Most tools use semantic signatures to identify well known pieces of malicious code (malware samples). Apart from the common problems associated with static analysis [6], when applied to IoT, this technique may present a disadvantage: malware samples have to be known. IoT devices may serve very different purposes. Malware can also be very diversified if developed with the device's functionality in mind possibly

[6]In the latest years several techniques of code obfuscation have appeared hindering the use of static analysis. This does not mean that this method does not serve its purpose: it does and it's widely used at the present time. However it does not work when the malware code is correctly obfuscated.

leading to a preference over the detection of zero day malware. The detection of this type of malware cannot be performed solely using static analysis: if the patterns to detect a piece of malware are not known it will not be detected.

Dynamic analysis.

Dynamic analysis or behavior analysis complements static analysis by monitoring events in the execution of a program and inferring its behavior. This has the advantage of being able to detect obfuscated malware— it's the execution that is being monitored, not the code— and even detecting zero day infections through detection of non-intended behaviors. However, two criteria should be taken in consideration when using this solution: the detection rate and the computational costs. Computational costs can be important since single board computers, despite the word "computer" in the name are more restricted than commonly used general purpose machines resulting in the need to evaluate the performance of several dynamic analysis techniques and detection models on such devices while still maintaining an adequate trade-off between detection rate and performance.

In summary, traditional malware detection does not function across all IoT devices, particularly restricted devices. Future solutions for this problem may lie somewhere between creating an energy profile for each equipment (executing malware plus the original code equals to executing more instructions and thus increasing power consumption), to the use of dedicated hardware units inferring the behavior of the code being executed or others. Meanwhile these topics are outside the scope of this work which has the intent of reinforcing that counterattacking threats is not a trivial issue and introduces new problems when applied to IoT.

At the same time it is also necessary to profile which gains could be obtained in infecting IoT devices in order to better understand the magnitude of the threat.

2.6.5 Possibility of IoT botnets

In a ecosystem like IoT, similarly to what happen nowadays in the computer world, the main menace may not come from script kiddies trying to jam the wireless signals neither from people trying to compromise devices just for fun or to test their skills. What is alarming is the possibility of automatic widespread of infections with financial gain as a goal. Currently, malware is heavily supported by an organized underground economy[15]. Such economy allows, among other things, people willing to commit cybercrimes paying for access to a set of infected machines— zombies— as well as tools and services designed to compromise more nodes. Revealing an expansion from computers to mobile devices, Android botnets are a reality at the present time— they are no more than a "business opportunity" foreseen by groups of individuals— leaving the question of how this economy will perceive the rise of IoT.

Possible services provided by IoT devices for the malware "industry" don't necessarily conflict with the ones offered by a computer or mobile device: there may not exist personal

information about individuals neither a SIM card easily used to obtain money,... Instead, the services a IoT device bids are dependent on the device itself since no particular use is imposed. Some devices, however, may be very attractive to certain groups such as industrial devices. The Stuxnet worm [7] and related malware proved to the world that cyber espionage and sabotage are desirable for some individuals, reinforcing once again the importance of designing secure systems.

Several examples could be provided considering different types of services and their location from hacktivism and cyberterrorism to examples of cyber espionage and sabotage. However, the greater advantage that the proliferation of malware may offer to its economy is the number of possibilities to explore. As for the counterattacking counterpart, one major issue arises: not knowing for sure what malware may target, difficulting the act of taking defensive measures.

2.6.6 A parallel between restricted devices and single-board computers

There is a clear differentiation between restricted devices and single board computers regarding not only capabilities but also in proposition to contain vulnerabilities, those same vulnerabilities being explored, detected and cleanse of non-intended behaviors.

While restricted devices usually don't have an operating system, single board computers have causing them to execute multiple programs/processes concurrently. Since the complexity level is higher, there is a greater chance of containing vulnerabilities. A single one of the running applications, containing vulnerabilities, may be exploited in order to compromise the device. On the other hand, malware detection can follow traditional approaches followed in computers.

In contrast, restricted devices with limited resources and without an operating system only execute a single piece of code, effectively limiting the possible location for vulnerabilities to a single "program". As a disadvantage, traditional malware detection is not adequate.

2.7 Security overhead

It's undeniable that securing the devices and its communications is imperative in order to guarantee something as basic as that the communications are being held with the device and not with an entity impersonating the device. This ability to trust in the device's messages is not, however, devoid of disadvantages. Executing additional cryptographic code in order to secure the communications uses:

Additional CPU time— Time that the CPU dedicates to the execution of cryptographic primitives instead of executing another piece of code;

[7] Stuxnet is a worm discovered in July of 2010 whose samples can be traced back from more than a year before (June 2009) which intended to sabotage industrial control systems. For that it spread through machines running Windows operating system until the final target were to be achieved: Programmable Logic Controllers (PLCs). This attempt of sabotage besides requiring real hardware to be tested -implying resources of some sort-, explored 4 zero day vulnerabilities, compromised 2 digital certificates, was capable of injecting code into a PLC and hiding itself after doing so[16].

Energy— Energy used to keep the device active and not in an inferior energy consumption mode plus energy used to operate the respective CPU units during algorithm's execution;

RAM— Temporary (or permanent if state is required) increased RAM usage;

Permanent memory— The size of the code increases with the addition of cryptographic primitives.

Adding security requires an adequate choice in both devices and primitives to use in order to assure the fulfillment of the device's purpose without being crippled by addition of security. Failing to do so has consequences such as:

- Inability of complying with real-time requirements;
- Requiring more power than has been dimensioned;
- Decrease of functionalities.

Irrefutably, security poses an overhead which manufacturers may or may not be willing to embrace, reinforcing the importance of choosing adequate cryptographic primitives to the importance of data and device capabilities.

2.8 Chapter overview

Security is not assured by the simple use of cryptographic methods but instead a combination of factors. Safe programming, physical protection, protocol design and cryptography must work together before the mentioning or assumption of security. The number of possible attacks and security threats to these tiny and fragile devices will continue to grow along with their interest in similarity of what has happened when introducing new technologies in the past such as Android.

From the set of fundamental principles needed for their security one issue will be further studied from a performance and requirements based perspective: what different cryptographic techniques and cryptography based solutions offer and their respective costs and requirements.

Chapter Three

Securing IoT

"Securing" might be a very deceptive word since it implies the existence of security. And security, as many other related properties, is a very delicate state. A state which can only be provided by assessing all the possible situations and scenarios and, in the end, might become impossible to prove. Security, as a chain, is only as strong as its weakest link consisting of a series of components which, when erroneous, will not verify this property. Nevertheless, its implementation has to germinate somewhere and starting by discussing communications security is a valid place to begin as any other and, in IoT, securing such communications might be more important than it appears initially.

3.1 Trusted computing base

References to the expression "Trusted Computing Base" can be traced back to 1979 [17] where it is described as the access control mechanisms for operating systems, referring only hardware and software mechanisms. Meanwhile, more recent definitions [18] evolved to consider also firmware defining TCB as:

> The totality of protection mechanisms within a computer (i.e., hardware, firmware, and software) responsible for enforcing security.(...)

When applied to IoT and WSNs, however, this definition could perhaps be extended accordingly to the point of view: if the set of devices/gateways that comprise the scenario is considered a tightly coupled system in the same way that peripherals inside a computer are, the TCB corresponds to the totality of protection mechanisms within the scenario itself. Such analogy might be reinforced arguing that a gateway receives data and sends control commands to the devices in a oddly resemblant way that a CPU/CPUs inside a computer do to peripherals. The major difference between this extended definition and the cited is that the resulting scenario of multiple small components interacting is not a computer, however it can still be considered as an entity to protect as a whole due to the risk that compromising a single entity might cause to the system as an all.

Protection of such entity must consider protecting also the interaction between its components which may be physically separated and communicating over a shared medium. Such can be performed through the use of cryptography which aids in providing a number of properties[19]:

- Authentication— verification of the message origin. Messages are originated at a known source and not at an external party;
- Authorization— the process of verifying access privileges[20]. Cryptography itself does not provide authorization, however, authentication is essential for implementing authorization mechanisms and authentication can be provided through it;
- Data confidentiality— act of hiding a message contents;
- Integrity— guarantee that communications were not altered while in transit. This is not exclusively applied to attacks but to all situations where messages, upon reaching the destination, deviate from the original content;
- Non-repudiation— the inability of a message sender latter denying sending it.

3.2 Sufficient and rational security

As mentioned previously, the act of securing the devices communications introduces an additional overhead. What would be desirable are fast, low memory, cryptographically strong algorithms that perform similarly across heterogeneous architectures. This does not, however, comply with the real world where:

- Cryptographically stronger algorithms are often associated with a lower throughput;
- A given implementation may favor execution time by neglecting memory (e.g. use of lookup tables, loop unrolling, ...) but there is always an obvious trade off;
- Performance of the same exact code can vary according to architecture characteristics such as the number and size of CPU registers or specific instructions.

However, it's important to realize that security mechanisms are no more than tools to enforce security policies. The recognition of this fact leads to the conclusion that a given cryptographic technique may be considered strong in one scenario while being considered weak in another and vice-versa: it simply depends on the level of security required by the security policy. This "sufficient security" can reveal itself to be a double-edged sword between resources required to apply cryptographic techniques and the security of those same techniques. Nonetheless, it implies that multiple methods should be considered when discussing the act of implementing security. Disadvantages in one scenario may not seem so in another

and advantages may prove to be irrelevant: only the security requirements may define it by setting the adequate security level— including communications security level.

Besides setting an adequate "cryptographic level", it's essential to choose wisely exactly what security properties should be verified. For instance, IoT shows a small nuance regarding data privacy that is worthy pointing out. In a computer the data consists in data fluxes triggered by humans— either by starting a program or performing another action— containing data related to a particular human action which ultimately are related with the human himself and obtaining/intercepting the data equals to obtaining information about such human. That's where data privacy should be applied assuring that the transmitted information and the knowledge of the action that led to such flux of information can only be obtained through the mentioned human. Meanwhile, in IoT sensors such action can be worthless under one circumstance: when sensors are deployed in a public access area. If data can be obtained through other (even perhaps easier) means than requesting it from the device, there is no data privacy despite the use of techniques to do so. Such paradox indicates that in likewise situations it should be clear if data is worth hiding in each and every scenario— and in public access scenarios this answer might be negative. This does not argue with the importance of data privacy in many if not most of situations but simply states that there is no point in applying cryptographic techniques to enforce it where it cannot be enforced by them indicating that deciding which security properties should be a rational process in order to apply rational security measures— cryptographic actions that effectively enforce security properties.

3.3 Symmetric Ciphers

Symmetric ciphers refer to ciphers which use the same key to perform encryption and decryption of, respectively, plaintext and ciphertext. Despite the existence of asymmetric cryptography, symmetric ciphers have been extensively used due to requiring less computing power and, consequently, being faster.

Such ciphers only require that a secret has been shared amongst parties— the encryption/decryption key.

3.3.1 Block ciphers

Block ciphers are algorithms that operate through fixed-length sets of bytes (blocks) mapping n-byte size blocks of plaintext into n-byte blocks of ciphertext where the algorithm mapping between plaintext and ciphertext is specified by a secret key K. Such situation is depicted in figure 3.1.

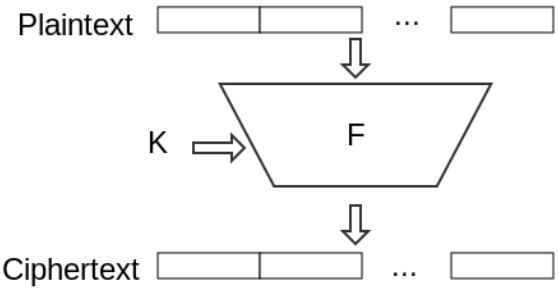


Figure 3.1: Operation scheme of block ciphers.

When simply mapping between plaintext and ciphertext, however, one situation occurs: equal plaintext block will originate the same ciphertext blocks. This simple transformation of text, without additional complexity, is called Electronic Codebook mode (ECB)— one of the block ciphers mode of operation— and it should be avoided specially in the context of IoT. Given that some of the original plaintext is known— perfectly reasonable knowing the message format/envelope used to transmit data— cryptanalysis to obtain the secret key may be possible. This problem had been identified long before IoT, however, it can provide an additional risk in such scenario: in devices providing sensor data, more correlations of plaintext-ciphertext can be inferred by both varying and sensing the surrounding environment (e.g. changing temperature near a temperature sensor). Besides the risk of cryptanalysis, using ECB mode also allows for replay attacks.

Operation Modes

Meanwhile there are several other modes for block ciphers that not the ECB mode which prevent such situation relying on the use of more factors than the mere shared key such as[21]:

- CBC (Cipher Block Chaining)— The first block of plaintext is xored with an Initialization Vector (IV) [1] before being ciphered. Consecutive blocks are xored with the previously obtained block of ciphertext before being ciphered. Decryption mirrors those operations in order to obtain the original plaintext.
- OFB (Output Feedback)— This mode transforms a block cipher into a stream cipher by the use of feedback. In the first block, the IV is ciphered, originating a keystream[2]. The resulting keystream is then xored with the plaintext in order to obtain the ciphertext. Meanwhile, n-bits of the keystream are reused as the IV for the next block, generating a new keystream.

[1] An IV is a fixed-size input to a cryptographic primitive used in order to increase the entropy level. Such values are typically required to be random or pseudorandom and non-repeatable (at least during a reasonable amount of time).

[2] Stream of random or pseudorandom values.

- CFB (Cipher Feedback)— Similarly to OFB, this mode also transforms a block cipher into a stream cipher. The difference between the two modes is that in CFB mode instead of the keystream, n-bits of the resulting cryptogram are reused as IV for the next block.

- CTR (Counter)— CTR mode distinguishes itself from the previously presented modes by not using resultant parts of the cryptographic primitive as IV bits for the next block. Instead, starting with a negotiated value *ctr*, each *i-th block* to cipher is xored with the resulting keystream of *ctr + i*. Although this mode converts a block cipher into a stream cipher, random block access is possible as in ECB since it's only necessary to know the block number and the initial *ctr* (besides the shared secret, obviously).

All presented modes, independently of transforming block ciphers into stream ciphers or not, assume state. This strongly implies that stream ciphers should also be considered and not discarded due to the need of keeping state: block cipher operation modes that do not reveal patterns also require state and considering them for use brings stream ciphers into the equation.

3.3.2 Stream ciphers

Stream ciphers are a practical approximation of a One-time pad[3]. Stream ciphers produce a continuous, pseudorandom keystream which can then be used combined (usually using the Xor operation) with the plaintext, producing ciphertext[23]. The shared secret, K, defines the behavior of the pseudorandom generator by altering its internal state. Representation of this behavior is depicted in figure 3.2.

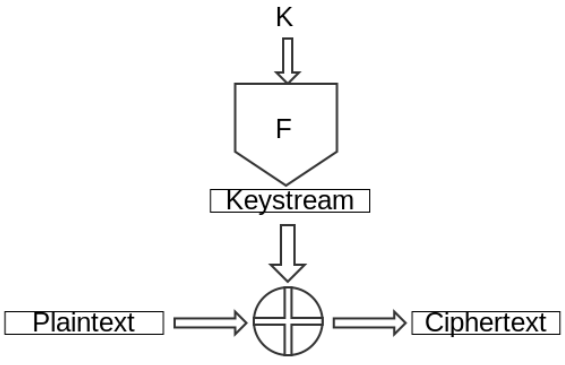


Figure 3.2: Operation scheme of stream ciphers.

It's important to note that for the same key and plaintext, without a previous state (new initializations), the resulting ciphertext will be the same. Therefore, if a stream cipher's

[3]One-time pad is a perfect, theoretical cipher presented by G. Vernam in 1926 where a truly random key whose length is equal or greater than the data is used to encrypt it. Since the key is completely random and at least of plaintext length, there is nothing that differentiates the ciphertext from truly random bits.[22]

internal state is restarted/reset multiple times, it is a good practice to combine the key with an IV in order to avoid replay attacks.

Since stream ciphers do not operate through block of data, one advantage is obvious: not requiring padding in order to cipher a given amount of plaintext as it may be necessary with block ciphers. Such property, in practice, means that less bytes may need to be processed by these ciphers.

3.3.3 What they provide?

Symmetric encryption, as well as all the encryption techniques, nullify eavesdropping assuring data privacy. If there is guarantee that the secret is only shared among parties, only the intended entities will be able to interpret the information exchanged. Simply encrypting messages, however, does not prove that the message has not been altered (no integrity checking) and does not prove that the message originated in the intended source (no non-repudiation). In order to provide both data privacy, message integrity and/or non-repudiation, encryption must be combined with additional cryptographic techniques.

This type of encryption can be applicable to scenarios with the most restricted devices. Assuming that keys are placed in devices (have been pre-distributed during manufacturing or other techniques), devices can then apply privacy to their communications using a single key. Symmetric cryptography is also known for being faster than alternatives being adequate to environments where throughput or power constrains apply (due to faster encryptions/decryptions, devices can dedicate more time to be in a low power mode). Meanwhile there is a clear scenario distinction between block and stream ciphers. Stream ciphers are known for being faster than block ciphers however this usually translates in more memory usage. Some stream ciphers are also known for being extremely lightweight however offering low data rates. Block ciphers, on the other hand, are more balanced.

3.4 Message Authentication Codes

Message authentication codes (MACs) are pieces of information resultant of an operation over a given message and a secret key with the intent of assuring integrity and authenticity. A MAC intent is the same as a checksum— prove that the message did not suffer changes— with the particularity that only entities with knowledge over the secret key can verify such condition. It does not provide non-repudiation since any subject knowing the key and the message could be the source not tying the message to a specific source, however, it authenticates the message— the message originated in a valid source (even if being replayed later, the message was in fact created by a valid source).

Due to the intent of providing integrity checking, one property of good MACs can immediately be extracted: small changes in the original text should be translated into significant MAC changes (same as in good checksums). Another important characteristic is that a MAC must resist existential forgery. That is, even if an attacker would have access to an oracle (containing the key used in the process) capable of generating MACs for given plaintexts, the

attacker still would not be able to extract the secret key[24]. The number of captured pairs of plaintext/MACs should not provide inference about the used key.

There are several strategies to produce such codes including[25]:

- Using a cipher in CBC mode— since CBC block encryption depend on the previous block, changes in blocks are propagated until the last block (used as MAC) being possible to verify the correctness of the message;
- Hash function based constructions— using a construction based on a shared key and the message, a hash is generated. Since hash function properties include[26]: (i) resistance to discovery of original text; (ii) resistance to discovery of a second text with the same hash; (iii) collision resistance, an attacker cannot recover the key neither can forge new messages;
- Using pseudorandom functions (XOR MACs)[27]— Using a pseudorandom function, the internal state is initialized by the secret key. The message, divided into blocks, is then processed by the pseudorandom function, generating a series of keystream blocks. Those blocks are then xored in order to obtain the MAC.

3.4.1 HMAC

HMACs or Hash-based message authentication codes deserve a special place along the hash based techniques to generate MACs due to the fact that it is extensively used including in both IPSec and TLS. First proposed by Bellare, Canetti, and Krawczyk[28] in 1996, the HMAC construction is based in two passes of a generic hash function, a secret key and padding as depicted in figure 3.3.

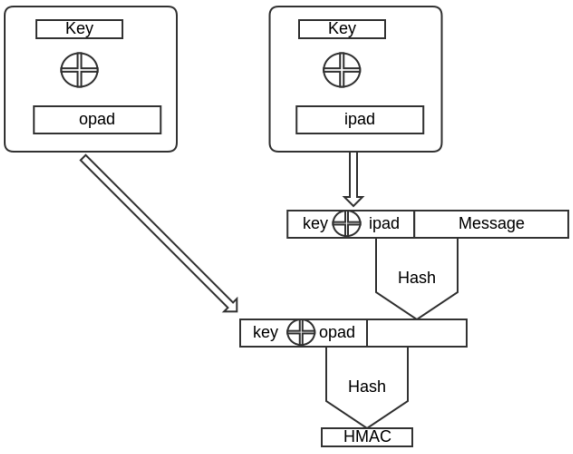


Figure 3.3: Scheme of the HMAC operation.

Construction strength depends on the underlying hash strength, however, besides increasing the time necessary to create rainbow tables, weak collision resistance may be masqueraded

due to padding addition and two hash passes[4]. For example, while the MD5 hash use is not recommended where collision-resistance is important, the HMAC construction using MD5 has yet to reveal a practical vulnerability[29].

Since calculating a hash usually requires more instructions (and therefore more time) than performing xor operations between the key and padding (and those can be precomputed), HMAC performance can be correlated with the underlying hash function performance, being the performance as varied as in the possible hash functions universe.

Similarly to the advantages provided by symmetric ciphers, MACs are the best choice for very constrained devices (again, assuming that keys have been pre-distributed). Symmetric cryptography is, in general, faster than the asymmetric counterpart and can be deployed in a larger number of devices. MACs should be used whenever the devices are restricted however their communications require data integrity and/or authentication and the property of non-repudiation is not relevant.

3.5 Public-key cryptography

Public-key cryptography or asymmetric cryptography, in contrast to symmetric, is not based on the existence of a shared key. Instead, a pair of distinct, mathematically correlated keys is used: a public key and a private key. The private key is inherent to a single entity (e.g. device) while public keys can be freely distributed since the mathematical correlation between keys cannot be solved in polynomial time not incurring in risk of derivation of the private key if the keys used are large enough. Data sent from a device to another can use the destination device public key to encrypt data with the guarantee that only the target device will be able to decrypt it. Such situation is depicted in figure 3.4.

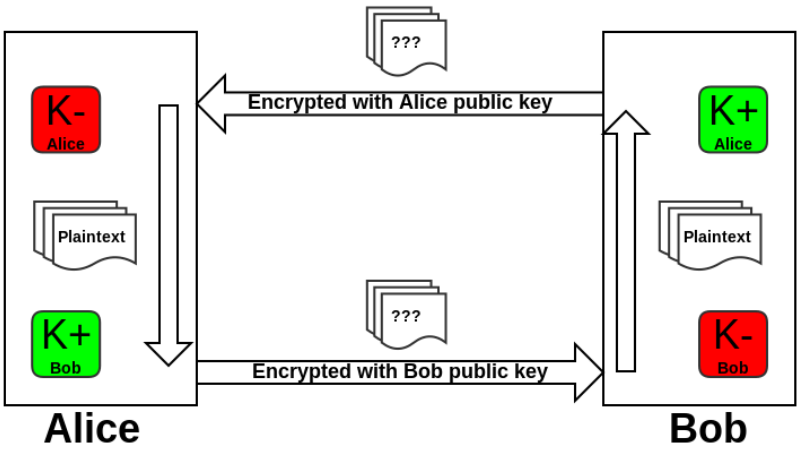


Figure 3.4: Asymmetric encryption and decryption scheme.

[4]Warning: this may also not happen. Careful planning is required when dealing with hash functions where collisions can be found by performing appropriate research about the possibility of collisions using HMAC with each particular considered function.

The illustrated situation seems to have an apparent problem. If Bob receives a message encrypted with his public key he can decrypt it using its private key. However, *apparently* he cannot be sure that the message originated in a valid source— Bob's public key is public and available to everyone including eavesdroppers. Meanwhile, this is not quite true. If the message origin decided to create (for example) an hash for the message and then encrypt it with its own private key, both authenticity and integrity of the message would be assured. Even more, the message origin could be pinpointed exactly, providing the property of non-repudiation. This is what is called a digital signature. Digital signatures are no more than pieces of information that, besides assuring integrity of a larger piece of information, allow the recognition of the entity that produced it.

As mentioned before, security in asymmetric techniques relies on the knowledge of the private key only by the proper entity. It's then crucial to ensure that the private key cannot be derived from the public key in polynomial time using an adequate key size and algorithms based on either: (i) Integer factorization problem; (ii) Discrete logarithms; (iii) Elliptic curves.

3.5.1 Elliptic curve cryptography

The use of public-key cryptography presents a crucial disadvantage over symmetric techniques: is several orders of magnitude slower[30]. Besides, while in symmetric algorithms as the key grows linearly with the processing speed, the same does not apply to asymmetric methods. That is, if not considering techniques based on elliptic curves (ECC). Such algorithms have already been described as ideal for embedded applications due to a decrease in both processing speed and memory requirements compared to integer factorization/discrete logarithms techniques due to a smaller key size to offer equivalent security. While solving other asymmetric techniques has an sub-exponential complexity, ECC mathematical problems solving is considered totally exponential[31]. Relations among key sizes (in bits) can be observed in table 3.1.

Symmetric	ECC	RSA/DH/DSA[5]
80	163	1024
128	283	3072
192	409	7680
256	571	15360

Table 3.1: Key sizes to provide equivalent security in different techniques.[32]

ECC is the *de facto* choice when considering the use of asymmetric techniques to secure restricted devices. Implementations consume less memory and perform much faster (and with difference increasing as key length increases) including on constrained 8 bit microprocessors[33]. Still it is slower than many available symmetric key techniques. This does not, by all means, intends to say that public-key cryptography should be discarded: simply that advantages and disadvantages should be considered when deciding between the two.

[5]Discrete logarithms/integer factorization based algorithms

As in the computer world, hybrid cryptosystems can be used. Public-key cryptography can be used, for instance, for key distribution (and afterwards being used symmetric algorithms) in order to avoid key overuse and solving the key distribution problem. After the initial key exchange devices would be able to execute simply symmetric algorithms being subjected to their power/throughput/memory requirements. Besides the hybrid approach solving key distribution while having the symmetric ciphers advantages afterwards, a larger amount of permanent memory is required. It should be noted that even in the computer world whenever periodic or peak and sporadic data transmission occurs, the sole use of asymmetric methods is exceedingly rare. In constrained or even powerful IoT devices (but with less capabilities than a regular computer) there is no reason for it to happen unless no-repudiation is a must.

3.6 Routing and cryptography

There are two distinct situations that may arise: when communication is performed directly between a device and a gateway (or simply between two devices) and when it's performed through the use of other nodes. As mentioned previously, only using cryptography is not enough to eliminate threats: careful protocol design also contributes to this. And any protocol designed with the intent of security as well as including routing will only consider messages that have a valid source (authenticated) implying that devices participating in the routing process must have a way to validate messages in order to decide if they are accepted or discarded. In practice this means that using symmetric cryptography at least one shared secret will be used and in asymmetric cryptography a pair of public/private keys per device (it wouldn't make sense to have a private key shared amongst devices since they wouldn't be considered private anymore).

Further expanding this logic means that situations can be comprehended somewhere between a single symmetric key shared by all the devices to a unique key/pair for each pair of nodes in the routing path. The first has an immediate disadvantage: success in compromising a single node and obtainment of its secret key equals to obtainment of the secret key used by all the remaining nodes in the routing process. The latter leads to a differentiated situation among symmetric and asymmetric cryptography:

- When using symmetric cryptography, if there is a single key shared among pairs, only the next hop in the routing path is able to verify the messages' authenticity. Therefore, it is necessary to create a new authentication tag for each hop in the path. This eventually uses more resources and introduces a greater delay than simply verifying the message's authenticity. The same can be applied to more than a single pair of devices (groups of devices) except that tags are newly generated when communicating among different groups.

- Using asymmetric cryptography, all nodes in the path can verify the messages' authenticity. Authenticity can be verified since only the public key is necessary. However, it

implies that each device contains other devices public keys, using memory resources.

Either way, secure routing is not an easy task and cannot be addressed trivially. How to address routing should also be suited to the security policies and the scenario, defining how serious it is for the routing structure to be compromised.

3.7 Operator coupled security

In some scenarios, there may exist the need for devices to communicate using a cellular mobile operator. By doing so, and using the operator's infrastructure, the device's communications will be subjected to the security provided by those same infrastructures using, for that purpose, a SIM card.

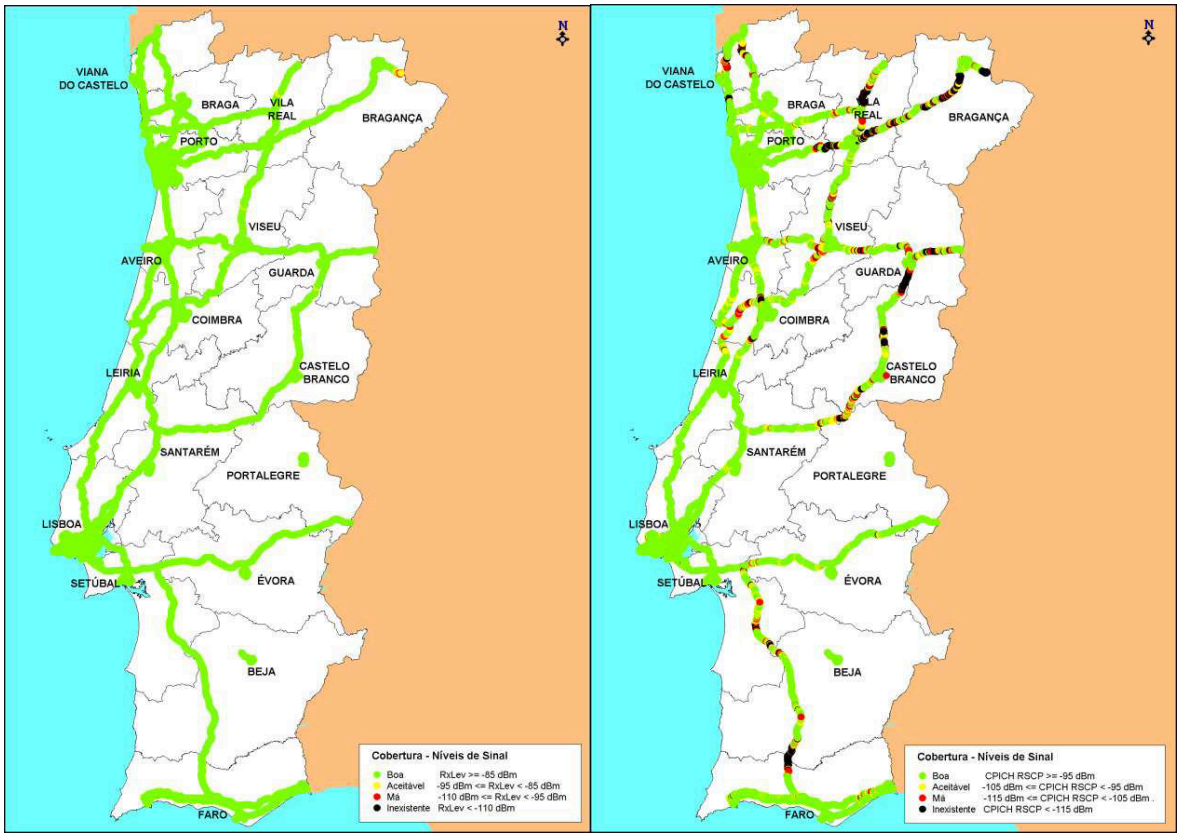
Securing devices using SIM cards implies a completely specified transport and security solution on its own since all negotiation, primitives to use, infrastructure and entity roles are predefined. Manufacturers can integrate GSM/UMTS/LTE modules into their devices and simply communicate using the operator already built infrastructure to transport data and operators wishing to infiltrate the M2M business can simply rely on their own and their partners infrastructure (for which use, contracts have already been previously established).

3.7.1 Wide availability

Along with the existing transport infrastructure, another great advantage appears: wide geographical availability provided by mobile technologies. The infrastructure is available in many remote areas. Oddly enough, this is where possible drawbacks begin to appear: GSM coverage is still wider than UMTS or LTE counterparts. Choosing to use a SIM card to secure devices is not a terminal choice— it's still necessary to choose which mobile technology will be used— and this may be conditioned specially in remote areas.

Using Portugal as a case study, a report from ANACOM [6] shows that mobile coverage is not evenly distributed amongst GSM and UMTS [34]. Figures 3.5a and 3.5b show the signal levels observed in the mentioned report for the Vodafone operator (the operator which presented less coverage).

[6] ANACOM or *Autoridade Nacional Comunicações* (National Communications Authority) is the entity legally responsible for regulation and supervision of both postal and electronic communications in Portugal.



(a) GSM coverage in Portugal[34]

(b) UMTS coverage in Portugal[34]

Figure 3.5: Comparison in coverage between GSM and UMTS in Portugal. Green indicates good signals levels, yellow acceptable, red bad and black nonexistent.

It's visible that there are multiple spots where UMTS signal is poor or inexistent. GSM however, despite containing areas with low signal levels, does not show spots with inexistent coverage. Also important to refer is that the study focused only on the main urban areas and highways: disparity in coverage may be even greater in more secluded areas. Despite LTE not being studied it's safe to assume that coverage is more restricted since it's a more recent technology and deployment began later.

The possible fallback to GSM use requires then a further study of this technology and its evolutions in a security perspective.

3.7.2 GSM

GSM or **G**lobal **S**ystem for **M**obile Communications is a standard accepted by ETSI in 1989 as the *de facto* standard for the second generation of mobile communications (2G), surpassing 2 billion users in 2006 [35]. Meanwhile, besides live, breathing users, GSM (and consequently GPRS [7]) have also been associated with M2M and IoT projects from industrial

[7]GPRS or General Packet Radio Service added packet capabilities to the original GSM, increasing possible throughput and expanding network capabilities.

data acquisition[36] to microbots[37] and others[38] with an expected almost exponential increase in subscriptions in the short-term future[39].

GSM security is granted by the SIM card present in a mobile terminal. Such card contains an IMSI (International Mobile Subscriber Identity)— a 64 bit number which identifies the equipment— and a secret 128 bit number, *Ki* (Authentication Key) along with hardware implementation of cryptographic primitives. The assumption that only the SIM card and another GSM system component inside the operators' infrastructure— the Auc or Authentication Center— know these numbers is the basis for GSM security, despite the fact that the IMSI is extracted from the card to identify the terminal every time it enters for the first time in a different geographical area (location area). There are also two additional components called the MSC or Mobile Switching Center and the VLR or Visitor Location Register which are responsible for negotiating security and storing credentials respectively. Whenever a mobile terminal intends to be authenticated before the network:

1. Without considering additional communication, messages eventually reach the AuC indicating the mobile terminal intention of being authenticated;
2. AuC generates a random number (RAND), calculates 2 additional numbers— SRES and Kc— using the mobile terminal Ki and algorithms known as A3 and A8 and sends them to the VLR;
3. The VLR sends only the RAND to the mobile station;
4. The mobile terminal calculates SRES and Kc since it also possesses the Ki and the A3 and A8 implementations;
5. SRES is sent to the VLR by the terminal;
6. If the SRES sent by the device equals to the SRES contained in the VLR, the authentication process is successful.
    a) Optionally, if encryption is desired, the Kc is passed to the BTS and used to encrypt future messages using the negotiated algorithm— A5/1 or A5/2.

For a clearer understanding of this process, entities involved in the GSM negotiation are depicted in figure 3.6.

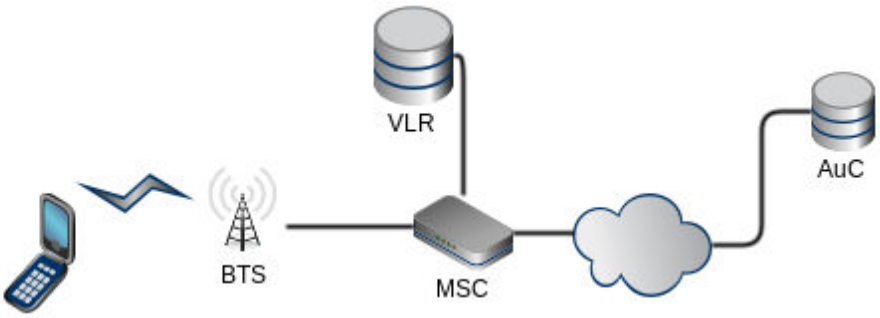


Figure 3.6: Very simplified scheme of the GSM infrastructure depicting entities involved in authentication.

Not depicted is the exchange of the IMSI for another number designated TMSI (Temporary Mobile Subscriber Identity). Instead of identifying the mobile using the IMSI, a TMSI is randomly assigned by the VLR to the mobile device with the purpose of anonymity.

From the described process, one of the first things to notice is that authentication is not mutual: only the terminal is authenticated. This, along with other problems have been identified in the GSM system since its conception. Toorani and Beheshti[40] did a great job at enumerating the 12 most important vulnerabilities in GSM, some of which are illustrated here:

Unilateral authentication. Already tackled in this work related to other issues, it allows man-in-the-middle attacks as long as the attacker possesses the equipment (fake BTS);

A3 and A8. Despite liberty to choose the algorithms, many operators used COMP128. Flaws discovered in this algorithm allow the exposure of the Ki;

Weak cryptographic algorithms. Both A5/1 and A5/2 allow the extraction of the encryption key in real-time;

Loss of anonymity. Despite the use of a TMSI instead of the IMSI, BTS impersonation can be used to reveal it;

SIM cloning. Obtaining the IMSI and the Ki is the only requirement to clone a SIM card— and both are possible to obtain;

Lack of end-to-end security. Encryption of data is only performed between the mobile station and the BTS.

Given these problems, the difficulty of an attack may reside mostly in obtaining equipment capable of BTS impersonation. Deploying physically unprotected devices is also increasingly threatening: while obtaining the Ki using the air interface takes hours, physical access to the SIM reduces that time to about one minute (not that it matters much if the devices will stay unprotected for days or more— any method will work).

It's also important to note that not all the presented problems are always true. Due to the exposure of GSM weaknesses, manufacturers deployed new authentication/key generation and encryption algorithms in more recent SIM cards even if intended to be used in GSM-only networks: A5/3 instead of A5/1 and A5/2 and COMP128-2/COMP128-3 instead of previously used COMP128. However relying in such updates may be a gamble— SIM cards can or cannot have updated cryptographic primitives, the only way to know for sure being testing each one of the SIM cards which may not always be viable. Meanwhile, the authentication process continues to be unilateral, allowing BTS impersonation.

While the previous presented encryption algorithms hold true for almost all data transmitted using GSM (including voice, SMS traffic and consequently WAP messages), there is one exception: GPRS. While the key obtainment for GPRS is exactly the same as in GSM (where only entity names change), GPRS uses a version of GEA (GPRS Encryption Algorithm) to encrypt all data. To this date, there have been not many significant studies regarding GPRS (and consequently, its enhanced version EDGE) security. However, one presentation at Chaos Communication Camp 2011 [8] shows that not all operators choose to encrypt their data. There is a probability of all data being transmitted in plaintext, being dependent on the operators' policies regarding GPRS security.

Even when GEA is used to encrypt messages one additional problem arises: the first two version of this algorithm hadn't been disclosed to this day. However, the first version of this algorithm (GEA/1), proved to be highly susceptible to attacks as it was possible to infer the internal state in minutes despite its obscurity at the previously mentioned presentation. Relying on security through obscurity has not proved successful in the past and GPRS may no longer have claims of security if the GEA family of algorithms looses its secrecy.

3.7.3 UMTS and LTE

In contrast with GSM, pure use of UMTS and LTE have not been exploited in a way that can compromise the mobile terminal. Starting with 3G (UMTS), new features include[41]:

- Mutual authentication: both the network and the mobile are authenticated before each other, eliminating the possibility of impersonation;
- Message integrity: not just encryption as in GSM;
- Use of stronger cryptographic primitives both for encryption and authentication/key generation.

Practically speaking, devices communicating using 3G and 4G cellular networks have a chance of being secure. There are, however, some nuances that must be taken in consideration:

[8] `http://events.ccc.de/camp/2011/Fahrplan/attachments/1868_110810.SRLabs-Camp-GRPS_Intercept.pdf`

- For devices capable of operating in dual mode (UMTS and GSM), a man-in-the-middle attack has been described. It's possible to impersonate a GSM BTS and force the hybrid device to communicate using the GSM scenario[42].
- More attacks based in GSM/UMTS interoperability are possible including eavesdropping (obtaining the encryption key) and device impersonation [43]— assuming that there can exist two subscribers with the same identity in the network.
- KASUMI, the encryption primitive used in UMTS and updated GSM SIM cards, despite using 128 bit keys, has cryptographic weaknesses. An attack described by Dunkelman, Keller, and Shamir[44] allows the obtainment of the key used with time complexity of 2^{32}. Despite this attack requiring known plaintext and related keys, not being practical in a UMTS scenario without flawed implementations, a decrease in complexity from 2^{128}(exhaustive search) to 2^{32} raises questions about the longevity of this cipher[45].

Despite those flaws, using pure UMTS or LTE communication eliminates the vulnerabilities presented by multi-mode operation and possible fallbacks to GSM authentication and the concerns about UMTS cipher longevity are only applied to the future. However, despite an increase in security in more recent mobile technologies than GSM, an additional issue needs to be addressed: where exactly data protection ends using operator coupled security.

3.7.4 End-to-end security

A very important security aspect is to identify exactly where secure messaging must exist in order to select the appropriate techniques. SIM card security can effectively secure communications between devices and the operator to which it is connected. However, when secure communication between devices and a server/machine located somewhere on the Internet is required, solely using SIM cards is not enough. Independently of the used cellular technology, eventually messages/data have to leave the operator's infrastructure and be routed through the Internet in order to reach their destination. Such situation is depicted in Figure 3.7.

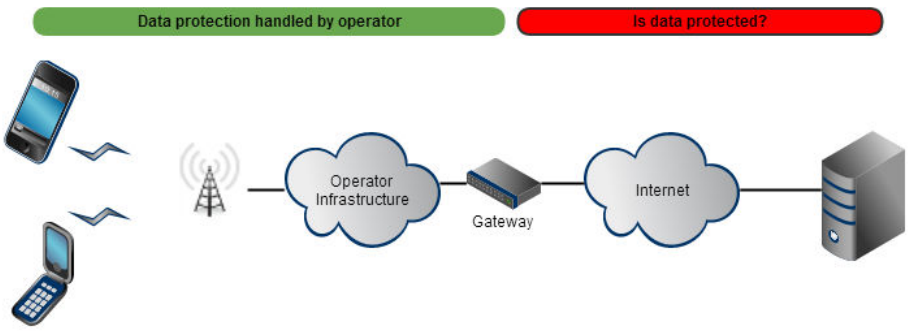


Figure 3.7: Representation of the data path using an operator's infrastructure. Despite the devices pictured being mobile phones, they should be interpreted as devices communicating using SIM cards.

Ultimately there are two questions to consider: how secure is the operator's infrastructure and how secure is the path between the infrastructure and the messages' destination. If an operator intends to provide M2M services, it can answer these questions since it knows its own infrastructure and will define the services it provides. Others may have difficulties answering such questions, requiring pondering on how important and sensitive is the data. When availability is important, however, the use of SIM cards may be nothing more than the mean to transport data, delegating security to other techniques which provide end-to-end secure messaging. This also does not consider scenarios where the operator provides security services (the operator decides the set of services it provides to end costumers) but a generic scenario where operators are no more than data pipes.

3.7.5 SIM malware

All operator provided security ultimately relies on the existence and integrity of the SIM card: the entity responsible for holding the shared secret which constitutes the base for secure communications. Meanwhile it happens that, despite all cellular protocol evolutions and security features, it may be possible to compromise the SIM independently of the cellular technology generation. The communication protocol is not the only requirement for security and the SIM functions as a safe anchor providing the devices a trusted module. Jeopardizing such role would invalidate the security provided by the SIM card.

However, at Black Hat USA 2013, a method for injection of malware into a SIM card was presented and was demonstrated through a proof of concept[46] by Karsten Nohl, resulting in the abolition of the concept of SIM as an impenetrable secure module.

Already mentioned previously, faulty or careless OTA implementations may lead to code injection. And it happens that SIM cards are one good example of such situation. OTA updates are performed through the use of a specially crafted SMS using, commonly, the DES 56-bit key cipher for protection. At the same time, it may be possible to send an OTA message with an invalid cryptographic checksum and still obtain a response containing a valid checksum created by the SIM card. Due to the relatively small[9] key space, it's possible to infer the OTA key, being of particular relevance the use of precomputed values (rainbow tables) to decrease the time complexity. From the moment that an attacker has the OTA key, it can deploy Java applications on the SIM card. Even further, it may be possible to bypass the "traditional" operations permitted by the Java environment being executed in the SIM in order to extract the Ki, alter the OS, etc. At least to perform simple operations (without Java environment bypassing) such as abusing payment schemes, only 3 requirements are to be met:

- The crafted SMS reaches the SIM card (it will not be handled by the device but by the SIM card itself);

[9]Specially considering the nowadays widespread use of, at least, 128-bit key primitives such as AES-128 to secure sensitive information.

- SIM card allows the issue of OTA commands protected using DES or a defected 3DES implementation (for instance using the same key for multiple DES operations);
- The SIM card replies to the SMS with a valid cryptographic checksum.

It's important to note that not all SIM cards are vulnerable to this attack (where one of the presented conditions is not met). However, such as the inclusion of resilient cryptographic primitives for use in GSM, it's a gamble. For geographical stationary IoT devices, operator filtering of OTA messages not originated in intended sources is enough to nullify this situation as long as BTS impersonation is not possible. Operators may also have patched this exploit through cryptographic primitive updating or assuring that the SIM card will not reply with a valid checksum. In fact, before disclosure of this attack several operators were notified in order to have time to mend it. Still, the ultimate question of which operators took effective measures remains. As with GSM cryptographic primitives, the only way to know for sure may be testing SIM cards individually, including in different network locations (OTA message filtering may only apply when devices are connected to the operator's and not one of the partners' infrastructure), which is not always viable.

3.8 Chapter overview

In this chapter an overview of cryptographic techniques and their properties have been discussed. A complete deployment scenario (use of SIM cards) also have been described announcing their advantages and disadvantages. The goal was not to completely describe all cryptographic options (for that it would require a book or more) but instead offer a perspective on how the most common cryptographic methods would apply to IoT. Secure routing was also mentioned as it is still a vast field for optimizations and presents a quite delicate balance between security and requirements.

After this general overview, it's time to observe exactly how these "constrained devices" behave while subjected to cryptographic algorithms to establish their impact using real values.

Chapter Four

Evaluating cryptographic implementations in IoT devices

In order to search for suitable algorithms for use in both restricted devices and heterogeneous environments, a small cryptographic library was created featuring three classes of algorithms: block ciphers, stream ciphers and HMACs. Such algorithms were then executed in multiple devices and evaluated according to several criteria.

4.1 Methodology

Results, by themselves and without providing sufficient context information, can prove to be utterly meaningless. Results as well as their conclusions are based on the idea that they will hold true in one or more situations. From the moment that this property is not verified, results become no more than mere abstract numbers. That is, if an experiment cannot be replicated under the same conditions or if consequent realizations of it are not consistent with the original data, the initial results and their respective conclusions are only applicable to the scope of the first experiment realization. Even without considering that computers and processing devices are mathematical tools, repeatability and reproducibility are essential for any experiment, in any field of study, intending to provide real data.

The methodology emerges as a key to ensure that the results are reproducible. Indicating all the conditions in which the experiment was conducted allows others to validate the results. Even more, by explaining why certain conditions were selected, it allows a more incisive discussion and others, with different goals/intents, can mold the conditions in order to satisfy their own agenda and compare to the initial results accordingly. A detailed methodology is essential since even with the same exact code, different results can be yield depending on the hardware, tools and their version.

4.1.1 Devices

Given the possible heterogeneity of IoT scenarios, cryptographic primitives were executed in multiple devices containing different CPU architectures.

The most restricted device used was a libelium Waspmote v1.1. Featuring an 8 bit ATmega1281 running at 8 MHz, 8 KB of SRAM and 128 KB of flash memory [47], this device has been advertised as adequate for WSN and IoT scenarios and is currently being used in several projects[1]. Besides the embedded flash memory this device also contains a slot intended for an SD card. However, such card is only used for storing data and files and not to contain code to be executed in the Waspmote. It is also important to state that this board is somewhat similar to some Arduino boards in terms of characteristics.

A DETPIC32 board was also used. This board was designed at University of Aveiro and features a 32 bit Microchip PIC32MX795 running at 40 MHz which includes 128 KB of SRAM and 512 KB of flash memory. Besides the difference in clock frequency and manufacturer, the fact that the microprocessor is classified as 32 bit can produce results that vary greatly from results obtained in the Waspmote.

Another device used was a Raspberry Pi Model B. This device almost doesn't require introductions: this single board computer high popularity made it the subject of several magazine articles, blog talks and open source projects. This popularity wages the Raspberry Pi as an interesting device to analyze performance due to the fact that it can be used in several enthusiast projects or even by companies intending to use the device's popularity as a mean to increase a given project popularity. The Raspberry Pi Model B features an ARM1176JZF-S CPU running at 700 MHz. With 512 MB of RAM and the permanent storage being in form of a SD card, this device has a much higher set of capabilities than the Waspmote or the DETPIC32 board, being perfectly reasonable its use as a gateway node.

In order to ease comparisons between devices, a parallel among them is presented at table 4.1.

	Waspmote	DETPIC32	Raspberry Pi
CPU	ATmega1281	PIC32MX795	ARM1176JZF-S
CPU Frequency	8 MHz	40 MHz	700 MHZ
Architecture type	RISC	RISC	RISC
Instruction Set	8 bit AVR	32 bit MIPS32	32 bit ARMv6
RAM	8 KB	128 KB	512 MB
Permanent memory	128 KB Flash + SD card	512 KB Flash	SD card
OS	None	None	Raspbian wheezy (Debian based)

Table 4.1: Comparison of the used devices

Choices in the devices tried to reflect the multiplicity of IoT nodes. 8 and 32 bit mi-

[1]libelium's web page includes a section entitled "50 Sensor Applications for a Smarter World" in which it describes 12 areas and more than 50 use cases to take advantage of data provided by sensors. For each area of application there are also a series of related articles some of which point to projects currently being developed. The mentioned website can be found at: `http://www.libelium.com/top_50_iot_sensor_applications_ranking/` (last visited on 27-10-2013).

croprocessors may yield different results specially if primitives were designed to operate (for example) between 32 bits of data at a time (being more adequate for this type of microprocessors). Besides differences in the clock frequency, changes in the manufacturer and instruction set are also reproduced in the resultant assembly code introducing variations in the code executed by the microprocessor.

4.1.2 Performance metrics

Throughput

In this context, throughput can simply be defined as the amount of data processed by a specific algorithm per unit of time. In other words, throughput can be expressed in any unit that correlates amount of data and time. To allow a broader range of possible future comparisons, 3 units will be indicated per algorithm:

Operations per second. The raw amount of operations that a device performs per second. This unit does not take in consideration the block size of block ciphers neither the amount of data to process passed to each call of a cryptographic function: it only indicates the number of successful returned calls from a cryptographic function.

Bytes per second The volume of bytes that are processed by a cryptographic algorithm per unit of time. For this reason, this is the value that should be taken in consideration when assessing if an algorithm is adequate for a given situation (when there are minimum data throughputs to comply with). It can be obtained merely by multiplying the operations per second with the amount of data passed to perform each operation.

Cycles per byte This efficiency measure indicates the number of clock cycles required by a device microprocessor to process each byte of data using a given algorithm. This is particularly useful when comparing different implementations of the same cryptographic primitive on the same device: if an implementation requires less cycles per byte than another, the implementation can provide a greater throughput (and vice versa). It also provides a view on the efficiency of the given algorithms on different devices and architectures. These values can easily be obtained by dividing the CPU frequency by the amount of bytes per second.

Throughput as an energy consumption metric

No actual measurements of real power consumption were held. While it is true that real measurements are the best way to determine power consumption, it is also true that time that a device has to spend being in active state to execute a cryptographic primitive is time that the device will not be in idle mode. Given the disparities in power consumption between active and idle states, it is possible to state with a certain degree of confidence that slower algorithms may consume more power (with the confidence level increasing along with execution time disparity), therefore making throughput a valid energy consumption measure.

There is also another variable to assess: if peripherals are active while data is being processed (e.g. a communications module while performing cryptographic operations on data before constructing a message). Peripherals active while waiting for a primitive to complete and in a power-saving state otherwise, increase the confidence level of throughput as a consumption measure.

Memory

Regardless of throughput and all its implications being a useful measurement, there are additional parameters that should be taken in consideration when selecting an adequate algorithm. One of such criteria is the amount of memory that it will require. Despite the fact that RAM has a smaller magnitude than permanent memory, both volatile and non-volatile memory usage should be taken into consideration since the depletion of either can bring forth undesired consequences.

The amount of permanent memory used by a cryptographic primitive is easy to quantify: it merely corresponds to the size of the binary file that will be written in the device's permanent memory or to the size of the executable file in devices with operating systems— being constant. The exact load in a device's RAM memory, however, is not so linear: different stages of an algorithm use different amounts of non-volatile memory. As so, the approach followed was to quantify the amount of RAM used as being the maximum amount of memory that a given algorithm uses which corresponds to the memory requirements to execute the primitive.

It's important to refer that comparisons cannot be established rashly both in permanent and volatile memory between devices with and without an operating system and even between different devices due to a multiplicity of factors:

- An executable file in an operating system contains an header specific to that OS, using more permanent memory space than it would without an header;
- Binary files to be written into devices may or may not contain different headers;
- Devices without an operating system do not use processes;
- Devices with operating system use processes and there is code, state and control data associated with each process, resulting in an increase of RAM usage;
- If a MMU is present, pages will be allocated instead of the "real", byte-exact memory required to run the primitive.

As such, even if the compiled code (assembly) to execute were to be exactly equal in both OS/non-OS devices (assuming same architecture), the amount of both permanent and volatile memory used would still differ.

Permanent memory could also vary in similar devices due to changes/existence of an header in the binary file. As for RAM, the existence/non-existence (and its use) of an MMU

can generate disparities as large as *page size - 1*, due to the need to reserve a memory page to store a single byte of information.

Differences between devices led to a disparity in RAM measurement. In the Raspberry Pi (device with operating system), a system tool was used to measure the total RAM used by the processes— *pmap*. In the Waspmote, since no tools to profile memory were available, the execution point where the call depth and memory allocation was superior was found and the amount of free RAM was measured using manufacturer's libraries. The DETPIC32, however, does not has, at the present time, libraries that quantify and handle memory usage like the Waspmote, being impossible to measure used memory without an implementation. Due to this, RAM was not quantified regarding this device.

A summary of the methods used to quantify memory usage is provided at table 4.2.

Memory type	Waspmote	DETPIC32	Raspberry Pi
Volatile	Manufacturer library	N/A	Process information
Permanent	Binary size	Binary size	Executable file size

Table 4.2: Comparison of methods used to measure memory in the devices.

It's also important to discriminate between base memory and cryptographic primitives memory. Any program, even if including only function prototypes, require a base amount of both permanent and RAM memory. The total memory used corresponds then to:

$$\text{Total memory} = \text{Base memory} + \text{Cryptographic primitive memory} \tag{4.1}$$

The base memory used during benchmarking doesn't correspond only to function prototypes but corresponds to the memory required by an *init* and *loop* functions— initialization and execution phases typically found in embedded systems— plus the code required to print debug information— corresponding to printing memory usage or information about the number of performed operations. Such code as used in the different devices can be observed in **Appendix A** and base memory values obtained can be observed in table 4.3. During exhibition of the memory results, both total and cryptographic primitive memory are presented since one corresponds to an increase in requirements by adding the primitive while the other the total memory requirements for executing and printing information about it.

Memory type	Waspmote	DETPIC32	Raspberry Pi
Volatile	688	N/A	1536 KB
Permanent	2670	4272	5610

Table 4.3: Base memory values in the devices.

In order to better comprehend the results, it also should be taken in consideration that, in the Raspberry Pi a complete operating system is running. RAM usage corresponds to the amount of memory required by a process to execute successfully and, as mentioned previously,

this also contains code resident in RAM. That resident code will be the primary source for RAM usage variations amongst primitives. This can be affirmed since no static variables neither dynamic memory were used leading to memory being reserved in the stack. However, in Linux, by default, 128 KB of stack space are allocated for each process (independently of being or not used). If the stack memory used does not exceed that value, no variations among data used by cryptographic primitives will be observed (and no implementations using amounts of RAM 16 times superior to the Waspmote's memory were selected).

Lastly but not less important, it should be noted that all the memory measures are presented in bytes, except when is specifically stated otherwise by indicating the unit.

4.1.3 Implementation properties

Obviously, any practical implementation contains a particular set of key characteristics and sometimes restrictions. This point explains the main features/properties of the implemented cryptographic algorithms.

Programming language. The chosen implementation language was C. Despite assembly languages being popular in optimized algorithm implementations, C code is somewhat portable among different platforms which does not happen using assembly code. Due to the intent of testing the exact same code against different architectures (given the heterogeneity of IoT devices), the use of assembly was discarded. Besides, this approach can be more approximate to the industry where a developed library may not be used in a single microprocessor/device but instead on a multiplicity of them.

Existing implementations. Instead of implementing each algorithm from scratch, existing implementations were modified to be platform agnostic. The immediate advantage of this approach is that a larger set of cryptographic primitives were tested. Implementing an algorithm from the specification can require a significant amount of time and is error prone. To maximize the number of implementations per unit of time this seems the more adequate course of action. Besides such implementation would be used only in this specific test scenario. Using modified existing implementations equals to benchmark implementations that are being used in practical scenarios, adding an increased level of practical applicability to the obtained results. Obviously, all the implementations were subjected to test vectors before and after the modifications to ensure correctness.

Platform agnostic. As mentioned previously, implementations were modified to be platform agnostic. This is quite possible starting by following one important MISRA C [2] advisory rule regarding types:

[2]Since C code is error prone and some embedded systems require stability and reliability over anything else, MISRA (Motor Industry Software Reliability Association), a collaboration between industrial partners, started seeking the best practices for developing such systems. As a result, MISRA C emerged as being both a subset and guidelines of the C language in order to comply with safety in embedded systems. Initially intended for use in automotive systems (as the name indicates) it has long expanded that use to being commonly associated with all types of critical systems.

6.3 (adv): 'typedefs' that indicate size and signedness should be used in place of the basic types.

Achieving this implies the discard of the common platform dependent integer types and instead using integer types where the size in bits is well known and does not vary among different compilers and architectures (called exact-width integer types in the C99 standard). This was implemented by the inclusion of a header file containing the definition of the types to use. Meaning that, to export the tested primitives for a different environment than the ones used, it is only necessary to modify the definition of the integer types in that header file.

Additionally, any eventual dependencies on external libraries or external components were also removed or replaced such as operations among blocks of memory (e.g. *memset*, *memcpy*,...) which became supported by an additional implementation instead of relying on system libraries. Following this approach led to platform agnostic implementations and the cryptographic code used in all of the devices is equal with the exception of the header file that defines the integer types size.

Data divided into blocks. Ideally data would be passed seamlessly to a given primitive and, at each predefined interval, it would be possible to obtain the exact quantity of processed data. In reality this does not happen: data must be passed in finite amounts— chunks of bytes— and the total processed bytes equals to the number of successful calls to the cryptographic function multiplied by the size of the byte chunks. While for block ciphers the pieces of data passed correspond to the block size of the cipher, the same could not possibly occur with stream ciphers and other cryptographic primitives (no concept of block). However, even with primitives that don't explicitly handle blocks of data, the data is actually processed in blocks. Varying the chosen block size could vary the obtained results slightly due to the overhead of multiple function calls/returns but this was not performed due to the belief that it would require large variations in block size (and therefore number of function calls) to alter the obtained results in a perceptible way.

For stream ciphers the data was passed into pieces of 80 bytes[3]. This value is significantly higher than the typical blocks of block ciphers, however, stream ciphers would be in disadvantage over these ciphers if a small size were to be chosen since their otherwise continuous operation would be constantly interrupted.

Meanwhile HMACs were benchmarked using 447-bit pieces of data. This may not seem logical at first glance, however, all underlying hash functions were Merkle–Damgård constructions. Such constructions operate on fixed size blocks and always add padding to accomplish a multiple of the block size, even when the data to process can already be divided into blocks (case in which a complete padding block is concatenated). At the same time, the chosen key

[3]This value was not choose at random. It actually corresponds to the sum of the minimum size of an IPv6 header (40 bytes), the minimum size of an UDP header (8 bytes), the minimum size of a CoAP header (4 bytes) and the size of the resource *coap://test/.well-known/core* (28 bytes) constituting a total of 80 bytes.

size for HMAC was 128 bits and, by chance, all the underlying hash functions operated in 512-bit blocks. This means that the chosen value of 447 bits equals to the maximum amount of data to perform an HMAC operation with a single underlying hash operation (only over one block) while having the minimum amount of padding possible (merely 1 bit).

4.1.4 Tools and compilers

Whenever available, manufacturer/device creator provided tools were used. The reason for doing so was to increase the level of realism in the approach and reproducibility in production scenarios. In most occasions, when tools to assist programming are available, programmers (including developers who work at companies) will use them. Even more, they might be encouraged to do so, as by using tools different than the ones provided by the devices manufacturers they might not have support or assistance from the manufacturer's side. If, by some reason, the code does not function as expected, accountability will lay upon programmers and project managers instead of manufacturers. [4]

Waspmote was the only device used for which a dedicated IDE was provided by the manufacturer in order to assist programming. Based on the Arduino IDE (asserting once more their similarities), this IDE is accountable for compiling code and uploading it to the Waspmote's flash memory. A more detailed look[48] identifies this tool as using the compiler *AVR-GCC*. Whether any optimization flags are used or not is (unfortunately) unknown. Compiler version, meanwhile, equals to the version installed in the system and, during benchmarking, the AVR-GCC version installed in the system corresponded to the version 4.7.2. It's also important to refer that the version 0.2 of the IDE was used along with the version 0.29 of the Waspmote's API[5].

DETPIC32 creators offer a tool in order to assist compiling for such device: *pcompile*. Further analysis reveals that this tool is not more than a Linux bash script which underneath resorts to using the *PIC32-GCC* compiler with the optimization flag *-O2*. Removal of such optimization flag was not performed. That would imply changing a script that the device creators implemented differing from the previously stated intended approach. It could be

[4]The libelium Waspmote is actually a good example of this behavior. In order to provide support for developers, libelium hosts an Internet forum where its representatives aid external developers in using both Waspmote and other libelium products. In such place, a topic was created by an user intending to use Makefiles and AVR-GCC directly instead of the provided IDE. Besides other responses it was specially stated by a libelium representative "However, your issues are out of our support (...)". Despite later helping users intending to discard the provided tools, it's visible that the first industry response is to differentiate between supported and unsupported situations.
Complete conversation is available at: `http://www.libelium.com/forum/viewtopic.php?f=16&t=8389` (last visited on 27-10-2013)

[5]The Waspmote API provided by libelium provides an abstraction layer when programming this device. It includes functions and data structures to interact with communication modules, expansion boards, Waspmote's integrated sensors and SD card and a number of other utilities including abstractions for communicating with a connected computer through a USB-Serial interface and a basic implementation of memory management. In this context, both communication (to report results) and memory management (for RAM measurement) from this API were used, hence the need to indicate the specific API version to assure that the results are reproducible.

argued that modifying an optimization flag would cause no harm. However, such approach would surely intend to leverage results obtained with optimization flags used and since it's unknown if the libelium Waspmote uses any optimization flags, there is no possible leverage. Such would only be possible if instead of using the Waspmote IDE, AVR-GCC were to be used directly and the optimization flags set manually (and therefore known, meaning that leverage would be possible). Since such approach is contradictory to the principle of using manufacturer tools, it was not performed. The compiler version used was the version 3.4.4.

Remembering that the Raspberry Pi features a entire Linux distribution, any choice of IDE's and compilers would be possible: there are no impositions from manufacturer tools. Therefore, the approach followed was to use the *GCC* compiler (version 4.6.3) without optimization flags. The use of no optimization flags might be controversial (specially after observing that the DETPIC32 code is indeed compiled with optimizations), however, this was done for a reason: this way it's possible to affirm that the obtained throughput is at least as fast as the results presented. Establishing a minimum boundary means that there is possibly plenty of room for optimizations in both throughput and binary size (GCC also features an optimization flag intending to reduce the generated code size) without even performing changes to the code but simply by altering compilation flags.

4.2 Block ciphers

Four block ciphers consisting of six variants were selected for benchmarking. The following pages present a brief description of such ciphers as well as the reasons for their choice, the best known attack and obtained results.

AES-128

No cryptographic library could be considered complete without featuring AES. Following a 5 year selection process, the block cipher Rijndael became known as AES (Advanced Encryption Standard) in 2001 with approval and standardization by NIST, being approved to provide data confidentiality to sensitive information in the US Federal Government (but not restricted for other uses)[49]. Since then, AES has been commonly deployed in hardware and software due to its unique combination of performance, security and flexibility. It's use was so widespread that nowadays, in regular computers, CPU manufacturers include extensions in their instruction set to perform AES operations, offering hardware implementations to increase throughput and mitigate side-channel attacks[50].

Accepting keys of 128, 192 and 256 bits and featuring 10, 12 and 14 rounds respectively, AES operates on 128 bit data blocks. The best published attacks revolved around side-channel attacks which do not compromise the cipher itself but instead, specific implementations. Still, cryptanalysis breaks for the total number of rounds in less attempts than brute force have been published, being the best authored by Bogdanov, Khovratovich, and Rechberger[51]. This attack, however, does not provide any real threat against deployment of full versions of AES-128 featuring a complexity of $2^{126.1}$, only slightly better than exhaustive search.

This particular implementation of AES-128 was based on an implementation by Texas Instruments[6] being adequate for embedded systems.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	620	7013	26400
Bytes/sec	9920	112208	422400
Cycles/byte	806	356	1657

Table 4.4: Throughput of AES-128

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1199	511	N/A	N/A	1548 KB	12 KB
Permanent	5392	2722	20552	16280	18881	13271

Table 4.5: Memory usage of AES-128

AES-256

Ironically, a better cryptanalysis for the 256 bit version of AES than for the 128 bit has been published with a data and time complexity of $2^{99.5}$. This attack, however, is a related-key attack [7] and, depending whether relations between keys can be uncovered or not, this attack may not apply. In situations where non-related keys are used, the best attack continues to be the one published by Bogdanov, Khovratovich, and Rechberger[51] (the same attack as in the 128 bit version) with a computational complexity of $2^{254.4}$ being only slightly better than exhaustive search.

Although neither of the attacks are computationally viable, when intending to use related keys, it should be preferable to use the 128 bit version of AES since it is certain to provide a greater throughput while offering better security (the related-key attack is not extensible to the 128 bit version).

AES-256 deployed version was based on a byte-oriented implementation by the Literate-code company[8]. Since AES-256 and AES-128 implementations differ greatly, comparisons shouldn't be established rashly amongst their results. The difference in requirements among implementations serves to illustrate the flexibility of AES.

[6] `http://www.ti.com/lit/an/slaa397a/slaa397a.pdf`

[7] Related-key attacks are a class of attacks where an attacker can observe the behavior of the cipher under different unknown cryptographic keys where a mathematical relation between keys is known to the attacker. Despite allowing for establishment of great cryptographic analyses and uncovering flaws, it may not be so trivial for the attacker to know such relation amongst keys (although in WEP this happened and led to practical exploitation).

[8] `http://www.literatecode.com/aes256`

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	401	3593	16812
Bytes/sec	6416	57488	268992
Cycles/byte	1247	696	2602

Table 4.6: Throughput of AES-256

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1113	425	N/A	N/A	1540 KB	4 KB
Permanent	4030	1360	13080	8808	12385	6775

Table 4.7: Memory usage of AES-256

Present

The Present cipher was chosen due to claims of being ultra-lightweight and suitable for RFID tags and sensor networks[52]. Featuring 64-bit blocks, 80 or 128-bit keys and a recommended number of rounds of 31, this cipher was benchmarked under 80-bit keys. To this date, only attacks for reduced versions of this cipher were presented.

It's also important to refer that the benchmarked version of Present, besides being intended for 8-bit microprocessors[9], uses a total of 1040 bytes of lookup tables. Different implementations, which wouldn't do so, would obtain lower RAM usages. Nevertheless, results for throughput and memory usage for this particular implementation can be observed, respectively, at tables 4.8 and 4.9.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	311	2602	23432
Bytes/sec	2488	20816	187456
Cycles/byte	3215	1922	3734

Table 4.8: Throughput of Present

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1783	1095	N/A	N/A	1540 KB	4 KB
Permanent	5714	3044	18244	13972	11341	5731

Table 4.9: Memory usage of Present

[9]Based on code found in: `http://cis.sjtu.edu.cn/index.php/Software_Implementation_of_Block_Cipher_PRESENT_for_8-Bit_Platforms`

RC5

RC5 or Rivest Cipher 5 is a cipher designed in 1994 whose block and key size as well as the number of rounds is variable. The choice of benchmarking RC5 laid upon its simplicity which is sure to reflect in low memory usage.

Parameter choice during benchmarking followed the original recommendation: 64-bit block size, 128-bit keys and 12 rounds. Due to RC5 operating on words which are half of the block size, the benchmarked version can also be referred to as RC5-32 and the best known attack for such mode (also considering the same number of rounds and key size) requires 2^{44} chosen plaintexts[53]. Such low complexity may be troublesome, delegating the responsibility of not allowing such attack to careful protocol design.

This particular implementation of RC5 is based on an implementation included in AVR-Crypto-Lib[10], being intended for embedded devices.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	1116,7	119759	357855
Bytes/sec	8933,6	958072	2862840
Cycles/byte	896	42	245

Table 4.10: Throughput of RC5

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	864	176	N/A	N/A	1540 KB	4 KB
Permanent	3724	1054	7056	2784	9040	3430

Table 4.11: Memory usage of RC5

XTEA

As in RC5, the choice of XTEA laid upon its simplicity. First described in 1997, this cipher features a 64-bit block size, 128-bit keys and the suggested number of rounds is 64. However, due to the non-existence of a cryptanalysis exploring the recommended number of rounds, XTEA was benchmarked using a total of 32 rounds (the nearest lower potency of two).

Without assuming weak keys, the best know cryptanalysis presents a complexity of $2^{126.44}$ XTEA encryptions and is valid to a number of 36 rounds[54]. Meanwhile, the existence of weak keys shortens the cipher lifespan with the best attack requiring 2^{62} plaintexts and $2^{31.94}$ XTEA encryptions and is valid to a number of 34 rounds[55]. It's also important to note that according to the same authors, the number of weak keys for XTEA is $2^{108.21}$, requiring careful planning on which keys to use to take advantage of the security provided by this cipher.

[10] `http://www.das-labor.org/wiki/AVR-Crypto-Lib/en`

XTEA implementation was based on an implementation provided by the PolarSSL library[11]. Due to the compact code size offered by this primitive in its original version, its code was expanded through the use of loop unrolling resulting in a second, optimized version which was also benchmarked. Results presented in tables 4.12 and 4.13 refer to values obtained using the non-optimized, original version of XTEA as well as the values for the optimized variant.

		Waspmote	DETPIC32	Raspberry Pi
XTEA	Operations/sec	770,25	35810	333310
	Bytes/sec	6162	286480	2666480
	Cycles/byte	1298	140	263
XTEA Opt.	Operations/sec	1210,2	71300,5	270540
	Bytes/sec	9681,6	570404	2164320
	Cycles/byte	826	70	323

Table 4.12: Throughput of XTEA

		Waspmote		DETPIC32		Raspberry Pi	
		Total	Real	Total	Real	Total	Real
XTEA	Volatile	766	78	N/A	N/A	1536 KB	0 KB
	Permanent	3786	1116	6864	2592	7632	2022
XTEA Opt.	Volatile	766	78	N/A	N/A	1548 KB	12 KB
	Permanent	14594	11924	17144	12872	19680	14070

Table 4.13: Memory usage of XTEA

4.3 Stream ciphers

Benchmarked stream ciphers featured the entire portfolio of the eSTREAM project with an additional two variants also submitted to this project. Emerging in the ECRYPT[12] project, eSTREAM objectives included the research of state-of-the-art stream ciphers featuring either high-speed or low resource consumption in hardware from a period comprehended between 2004 and 2008[56]. Currently 7 primitives have been selected: 4 software and 3 hardware profile ciphers. The choice of benchmarking such recent ciphers was due to the fact that benchmarking eSTREAM portfolio ciphers equals to benchmarking the current state of the art in stream ciphers[57] developed in Europe. Despite not being used at large scale nowadays,

[11] `https://polarssl.org/`

[12] ECRYPT or European Network of Excellence in Cryptology was a research effort funded by the European Commission in order to, among other things, provide state of the art advances in cryptography.

in the future their adoption whereas stream ciphers are intended may overthrow the use of more matured stream ciphers being of relevance to study how they behave when applied to IoT devices.

It's worth noting that all the benchmarked versions had as base code the C sources available at eSTREAM project's web page[13].

Stream ciphers act in the same way than a pseudorandom number generated defined by an internal state. Due to need of initializing such internal state, stream cipher initialization does not occur instantly. Results presented in this main section are results benchmarked after the initialization of stream ciphers. Meanwhile, benchmarking was also performed assuming repeated cycles of initialization/ciphering. Such results and correlation among the ones presented here can be observed at **Appendix B**.

Grain-128

Stream cipher Grain has underwent several versions since its conception. Due to the initial version being compromised, the original Grain suffered changes, leading to the Grain v1 specification. This specification consists of two ciphers: a version featuring 80-bit keys and 64-bit IVs and another accepting 128-bit keys and 96-bit IVs.

The choice of benchmarking the 128-bit version laid upon the fact that ciphers using 128-bit keys are (usually) more popular than ciphers using 80-bit. Meanwhile, this version, in contrast to the smaller version, contains known vulnerabilities which inspired a cryptanalysis better than brute-force by Dinur and Shamir[58] where a set of weak keys was found to reduce exhaustive search by a factor of 2^{15} (leading to the complexity being reduced from 2^{128} to 2^{113}). Still, even considering such attack, Grain-128 continues to offer more security than the 80-bit version where the key space is limited to 2^{80} keys.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	2,4	49,25	163
Bytes/sec	192	3940	13040
Cycles/byte	41667	10152	53681

Table 4.14: Throughput of Grain-128

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1174	486	N/A	N/A	1536 KB	0 KB
Permanent	4084	1414	8280	4008	8812	3202

Table 4.15: Memory usage of Grain-128

[13] `http://www.ecrypt.eu.org/stream/`

HC-128

This software efficient stream cipher operates using 128-bit keys and IVs and, at the present time, no critical flaws were uncovered with best cryptanalyses requiring more effort to perform distinguishing attacks[14] than to brute-force the key[59].

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	349,25	37313	125738
Bytes/sec	27940	2985040	10059040
Cycles/byte	286	13,4	70

Table 4.16: Throughput of HC-128

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	5354	4666	N/A	N/A	1552 KB	16 KB
Permanent	27562	24892	24016	19744	25313	19703

Table 4.17: Memory usage of HC-128

HC-256

Stream cipher HC-256 does not constitute part of the eSTREAM final portfolio. Nonetheless and despite HC-128 not known critical flaws, it offers a more secure alternative to it using 256-bit keys and IVs. Since some environments may impose the use of 256-bit keys, this cipher (also not presenting menacing cryptanalyses) is worth benchmarking.

It's important to note that execution of this cipher proved impossible in the libelium Waspmote due to the lack of RAM memory. Despite this, indication of permanent memory required could still be presented since it corresponds to the binary file size. The capability of uploading the code to the device is different from the device's ability of executing it and such value may serve as a guideline for use on similar devices (obviously with more RAM).

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	N/A	25773	81640
Bytes/sec	N/A	2061840	6531200
Cycles/byte	N/A	19,4	107

Table 4.18: Throughput of HC-256

[14]Distinguish between encrypted data and purely random data

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	N/A	N/A	N/A	N/A	1560 KB	24 KB
Permanent	31678	29008	27580	23308	29985	24375

Table 4.19: Memory usage of HC-256

MICKEY 2.0

This hardware profile stream cipher operates using 80-bit keys and IVs up to 80 bits. As claimed by the eSTREAM committee, no attacks other than side-channels attacks are known at the present day.[15]

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	9,25	333	1641
Bytes/sec	740	26640	131280
Cycles/byte	10811	1502	5332

Table 4.20: Throughput of MICKEY 2.0

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1088	400	N/A	N/A	1540 KB	4 KB
Permanent	5992	3322	9708	5436	9389	3779

Table 4.21: Memory usage of MICKEY 2.0

MICKEY-128 2.0

MICKEY-128 2.0 does not constitute part of the final eSTREAM portfolio. Nonetheless, this cipher is a more security robust implementation of MICKEY 2.0 accepting 128-bit keys and IVs up to 128 bits hence the reason for benchmarking it. As with "regular" MICKEY 2.0, no attacks with less complexity than brute-force (except implementation related) are available.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	6,5	287	1392
Bytes/sec	520	22960	111360
Cycles/byte	15385	1742	6286

Table 4.22: Throughput of MICKEY-128 2.0

[15] `http://www.ecrypt.eu.org/stream/e2-mickey.html`

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1146	458	N/A	N/A	1540 KB	4 KB
Permanent	6584	3914	40392	36120	9759	4149

Table 4.23: Memory usage of MICKEY-128 2.0

Rabbit

The software profile cipher Rabbit features 128-bit keys and 64-bit IVs and to the date, no real attacks have emerged in the performed cryptanalyses[16].

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	242,7	18578,5	82067
Bytes/sec	19416	1486280	6565360
Cycles/byte	412	27	107

Table 4.24: Throughput of Rabbit

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1170	482	N/A	N/A	1540 KB	4 KB
Permanent	6376	3706	11324	7052	10350	4740

Table 4.25: Memory usage of Rabbit

Salsa20/12

Software efficient Salsa20/r is an adaptable stream cipher where r defines the number of internal iterations, being possible to achieve a compromise between security and performance. The decision of benchmarking the 12 iteration version (Salsa20/12) was due to being suggested by the eSTREAM committee as being the version offering the best balance[17]. Besides the number of iterations, Salsa20 also accepts keys of 128 or 256 bits as well as an IV of 64 bits, being the benchmark results obtained with 256-bit keys. It's also important to refer that at the present time no attack with less complexity than brute-force is available for Salsa20/12.

[16] A key-recovery attack has been described by Lu, Wang, and Ling[60] however it assumes that relations between internal states are known being just a theoretical attack.

[17] `http://www.ecrypt.eu.org/stream/e2-salsa20.html`

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	71,14	12886,5	66352
Bytes/sec	5691,2	1030920	5308160
Cycles/byte	1406	38,8	132

Table 4.26: Throughput of Salsa20/12

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1234	546	N/A	N/A	1536 KB	0 KB
Permanent	8086	5416	8728	4456	8915	3305

Table 4.27: Memory usage of Salsa20/12

SOSEMANUK

SOSEMANUK is a software profile cipher accepting keys in any range between 128 and 256 bits and IVs with 128 bits with the particularity of, contrarily to many other ciphers who also accept multiple key sizes, only proclaiming 128-bit security. Due to such claim, this cipher was benchmarked under the use of 128-bit keys. Despite being subjected to several cryptanalyses, currently there are no attacks with complexity inferior to the claimed security and, therefore, for the benchmarked key value, the least attack complexity is obtained through the use of brute-force.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	593,8	20366,3	80316
Bytes/sec	47504	1629304	6425280
Cycles/byte	168	24,55	109

Table 4.28: Throughput of SOSEMANUK

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	3660	2972	N/A	N/A	1560 KB	24 KB
Permanent	49106	46436	58448	54176	35843	30233

Table 4.29: Memory usage of SOSEMANUK

Trivium

Featuring 80-bit keys and IVs, the hardware efficient cipher Trivium, at the present time, does not contain known flaws that allow its exploitation.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	59	10532	97641
Bytes/sec	4720	842560	7811280
Cycles/byte	1695	47,5	90

Table 4.30: Throughput of Trivium

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1089	401	N/A	N/A	1544 KB	8 KB
Permanent	18254	15584	25476	21204	15038	9428

Table 4.31: Memory usage of Trivium

4.4 HMACs

After benchmarking techniques providing data privacy, it's time to benchmark HMACs which in contrast allow enforcement of data integrity and authentication. For that purpose a total of 4 HMACs and 5 variants were evaluated.

Before presenting results it's important to recall that the HMAC construction compensates for some weakly collision-resistance hashes. In fact, none of the benchmarked HMACs has an associated cryptanalysis allowing key recovery neither other practical attack for finding collisions. However, despite the current existence of attacks, as cryptanalyses tends to improve instead of devolving, HMACs constructed over hashes containing vulnerabilities [18] may have a smaller longevity. Therefore, security considerations will be assessed to the underlying hash function instead of the specific HMAC.

MD5

MD5 was selected as an underlying hash function for one very simple reason: speed. Despite several recommendations against its use and being possible to obtain collisions with a time complexity of $2^{20.96}$[61], MD5 continues to be widely deployed due to its superior throughput. Meanwhile, as previously stated, the HMAC-MD5 construction is not subjected to the same vulnerabilities as MD5. Despite it, the small digest size (128 bits) may be preoccupying when long-term implementations are required.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	106,85	7262	33461
Bytes/sec	5970	405764	1869633
Cycles/byte	1340	99	374

Table 4.32: Throughput of HMAC-MD5

[18] More precisely, hashes who do not behave like pseudorandom functions.

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1148	460	N/A	N/A	1540 KB	4 KB
Permanent	11606	8936	15432	11160	11629	6019

Table 4.33: Memory usage of HMAC-MD5

RIPEMD-160

RIPEMD-160 deserves a highlight among hash functions. Appearing in 1996 as a strengthened version of the original RIPEMD, there aren't, after 17 years, cryptanalyses exploring its full version (although several exist for reduced versions) and it's unlikely that such will appear in the near future[62]. This long longevity was the reason behind this hash function being selected for benchmarking. As the name indicates, RIPEMD-160 produces 160-bit digests.

The benchmarked version of RIPEMD-160 was deployed based on an implementation included in the Cryptokit library[19].

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	23,6	3413	17304
Bytes/sec	1318,7	190701	966861
Cycles/byte	6061	210	724

Table 4.34: Throughput of HMAC-RIPEMD-160

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1318	630	N/A	N/A	1544 KB	8 KB
Permanent	32674	30004	26540	22268	15681	10071

Table 4.35: Memory usage of HMAC-RIPEMD-160

SHA-1

SHA-1 is a hash function developed by NIST in 1995 featuring a digest size of 160 bits. Since its inception it has been widely deployed hence the reason for its benchmarking. Meanwhile, cryptanalysis reveals that collisions can be found with complexity 2^{61}[63].

Due to its relevance and role in nowadays computers, two versions of SHA-1 were deployed being the first an implementation based on the one presented by AVR-Crypto-Lib and the second a throughput optimized version based on the PolarSSL implementation.

[19] `http://forge.ocamlcore.org/projects/cryptokit/`

		Waspmote	DETPIC32	Raspberry Pi
HMAC SHA-1	Operations/sec	24,5	573,6	2296
	Bytes/sec	1369	32050	128289
	Cycles/byte	5844	1248	5456
HMAC SHA-1 Opt.	Operations/sec	40,5	5303	20985
	Bytes/sec	2263	296305	1172537
	Cycles/byte	3535	135	597

Table 4.36: Throughput of HMAC-SHA-1

		Waspmote		DETPIC32		Raspberry Pi	
		Total	Real	Total	Real	Total	Real
HMAC SHA-1	Volatile	1166	478	N/A	N/A	1540 KB	4 KB
	Permanent	5304	2634	11704	7432	10515	4905
HMAC SHA-1 Opt.	Volatile	1308	620	N/A	N/A	1548 KB	12 KB
	Permanent	25780	23110	22816	18544	19774	14164

Table 4.37: Memory usage of HMAC-SHA-1

SHA-256

Being part of the SHA-2 set, SHA-256 produces digests of 256-bit size. Since its inception, this successor to SHA-1 has been extensively deployed in multiple scenarios justifying its choice as worthy of benchmarking. It's also important to state that, at the present time, no attacks were published for its full version but merely for reduced versions.

Adapted (and benchmarked) code for this primitive had as base code a SHA-256 implementation included in the AVR-Crypto-Lib.

	Waspmote	DETPIC32	Raspberry Pi
Operations/sec	10,9	482,5	1744
Bytes/sec	609	26960	97446
Cycles/byte	13136	1484	7183

Table 4.38: Throughput of HMAC-SHA-256

	Waspmote		DETPIC32		Raspberry Pi	
	Total	Real	Total	Real	Total	Real
Volatile	1566	878	N/A	N/A	1540 KB	4 KB
Permanent	6242	3572	12136	7864	10866	5256

Table 4.39: Memory usage of HMAC-SHA-256

Results show first and foremost attainable cryptographic security in IoT. Waspmote, the more constrained device, can cipher data at almost 50 thousand bytes per second and it's also possible to provide data confidentiality with an impact of less than 1% in both RAM and permanent memory.

DETPIC32 and the Raspberry Pi, on the other hand, are capable of obtaining values surpassing 2,5 MB/s and almost reaching 10 MB/s, respectively, in ciphering. Again, remembering that such values represent a lower boundary for the Raspberry Pi.

An additional comparison between the complexity provided by the benchmarked ciphers and respective resource usage can be observed in **Appendix C**, granting an additional perspective about the benchmarked ciphers being discussed in this section.

Raspberry Pi considerations

As expected, RAM usage in the Raspberry is directly correlated with the executable file size. Primitives containing more code use more volatile memory due to the situation predicted in the beginning, regarding processes containing code to execute. Variations in RAM values, at most, are in the orders of 24 KB considering an initial base value of 1536 KB representing an increase of 1.6%. Practically speaking, this device includes 512 MB making such fluctuation less than 0,005% of the device's memory usage.

Permanent memory, at a base value of 5610 bytes, increases between 2022 and 30233 bytes representing a maximum increase of over 6 times in the executable size. However, speaking in absolute values, this represents a maximum file size of about 30KB including base code. Such value is of little relevance since the Raspberry features an SD card and SD cards are sold in sizes of gigabyte orders.

Given the obtained values, is of little to no practical relevance to discuss both memory impacts on this device. This would be different if results caused a visible impact in memory (which they don't).

Throughput values, however, are relevant since (assuming there are no limitations in the communication rates) define the data capabilities and, if in use in form of a gateway, exactly how many nodes it can serve simultaneously.

As expected, differences among both manufacturers and CPU architectures, yields different throughput and memory magnitudes for algorithms among different devices. Meanwhile there are algorithms that, despite their ranking number offer some consistent results when transposed to different devices.

Starting with the Waspmote, SOSEMANUK and HC-128 perform exceedingly well in throughput while performing at bottom in both RAM and permanent memory. Meanwhile, there are other contestants worthy indicating as presented in table 4.40.

Rabbit reveals itself as a well-balanced cipher in terms of requirements: offers high throughput at medium requirements in volatile and permanent memory. AES reveals its

	Throughput		RAM		Permanent Mem.	
1	SOSEMANUK	47,5k	XTEA	0,95%	RC5	0,80%
2	HC-128	27,9k	XTEA (opt.)	0,95%	XTEA	0,85%
3	Rabbit	19,4k	RC5	2,15%	AES-256	1,04%
4	AES-128	9,9k	MICKEY 2.0	4,88%	Grain-128	1,08%
5	XTEA (opt.)	9,7k	Trivium	4,895%	HMAC-SHA-1	2,01%

Table 4.40: Top 5 performers in the Waspmote device

known adaptability to different scenarios with the 128 bit version being the block cipher offering best throughput and the 256 bit version deserving the third place in the permanent memory ranking (but lower throughput than the 128 counterpart). It's important to note that differences amongst different AES versions are not only due to the key/construction size but also related to being two distinct implementations. AES is well recognized for the ability to originate different implementation favoring throughput or memory accordingly to demand for either.

The, not widely adopted in the current Internet, block cipher XTEA proves to be highly flexible in constrained memory environments offering at least half memory usage than remaining contestants and nearly the same use of permanent memory. It's also worth noting that the optimized version of this cipher was obtained by loop unrolling and different levels of loop unrolling could be used to offer a balance between throughput and storage (since the RAM memory usage will continue to be equal).

Also interesting are the results scored by RC5. Featuring first place in storage use and second place in volatile memory among different cryptographic primitives, despite not appearing in the top 5, features the 6th place in throughput in contrast with the non optimized version of XTEA which features 8th.

It should also be noted that HMAC-SHA-1 was the only non-cipher to appear on the top 5 due to low permanent memory usage. However, it loses to HMAC-MD5 in throughput (by a factor above 4) and RAM use. When selecting between primitives this may prove to be a disadvantage towards the more preventive use of SHA-1 constructions instead of MD5 based.

MICKEY 2.0 and Trivium analogous to Grain-128, also appearing in the top 5 due to RAM and storage respectively, have a common problem: least throughput than the remaining options.

The DETPIC32 continues the Waspmote's legacy by also featuring both the HC family, SOSEMANUK and Rabbit as top throughput performers and also including XTEA, RC5 and GRAIN-128 in its top 5. Such situation is demonstrated in table 4.41.

In the meantime a new contestant steps forward: Salsa20/12. Such algorithm is present in the top 5 for both throughput and storage and also, deserving a spotlight due to being the only primitive to do so in the DETPIC32.

MICKEY 2.0 appears in the top 5 not due to RAM usage as in the Waspmote but

	Throughput		Permanent Mem.	
1	HC-128	298,5k	XTEA	0,49%
2	HC-256	206,1k	RC5	0,53%
3	SOSEMANUK	162,9k	Grain-128	0,76%
4	Rabbit	148,6k	Salsa20/12	0,85%
5	Salsa20/12	103k	MICKEY 2.0	1,04%

Table 4.41: Top 5 performers in the DETPIC32 device

due to storage use. Remembering that the DETPIC32 does not allow for volatile memory measurements, it would be interesting to see how this primitive behaves in RAM usage in this device. However, it also presents a problem: it's one of the slowest primitives (4th slowest) in this device.

Previously noted primitive RC5, despite not appearing in the top 5, maintains its position according to throughput appearing in the 6th place. This is relevant due to maintaining its appearance in the top regarding memory. Again, benchmarking RAM would be valuable since this primitive also deserved a significant place among RAM usage in the Waspmote. XTEA, in contrast, features the 11th throughput place being below average. Meanwhile, XTEA optimized version offers greater throughput (being at 8th place) but presenting one of the largest permanent memory usage (5th largest).

AES, in this device, loses its place as the fastest block cipher with AES-128 appearing quite below as how it appeared in the Waspmote: 13th in contrast with previous 4th. The spot it deserved regarding permanent memory usage also disappears with both versions being of average usage. These average results, associated with the high flexibility of this cipher, leaves room to imagine that it would be possible to increase either throughput/storage ranking compromising the other component.

Raspberry Pi top 5 performers are merely a recombination of the DETPIC32 performers with only one change: Salsa20/12 has disappeared from the top throughputs (moving to 6th place) and Trivium (which was placed 7th regarding throughput in the previous device) is included. Meanwhile, permanent memory usage top 5 is merely a permutation of the DETPIC32 top 5 results. Such situation is demonstrated in table 4.42.

	Throughput		Permanent Mem.	
1	HC-128	10059k	XTEA	2022 bytes
2	Trivium	7811k	Grain-128	3202 bytes
3	Rabbit	6565k	Salsa20/12	3305 bytes
4	HC-256	6531k	RC5	3430 bytes
5	SOSEMANUK	6425k	MICKEY 2.0	3779 bytes

Table 4.42: Top 5 performers in the Raspberry Pi device

Worth noting is that RC5 and XTEA, despite the usual appearance in memory usage top, also appear at 7th and 8th place regarding throughput with oddly, the optimized version of XTEA appearing lower than the non-optimized version but still above AES. Again, no attempt was made to optimize the code against a particular architecture and an architecture-optimized version of AES could possibly perform with much higher throughput.

HMAC limitations

From the primitives discussed so far, only a single HMAC has been mentioned. This happens because evaluated HMACs do not perform at the same rank as the other contestants except for HMAC-SHA-1 for permanent memory and running on the Waspmote. This poses a real limitation since while ciphers provide a security property (data confidentiality), HMACs indeed provide two— authentication and integrity. While the benefits of data concealing are clear, so are the benefits provided by HMAC and, in some situations, may even be the only relevant properties.

Meanwhile, as expected, HMAC-MD5 is the HMAC construction offering best results both in throughput and RAM usage in all the devices. As already mentioned, HMAC performance depends on the underlying hash construction and device's results validate what happens in the general computing world where MD5 continues to be widely deployed, against several recommendations, due to its high performance (although, as also mentioned, HMAC-MD5 does not present the same flaws). At the same time, implementations of HMAC-SHA-1 (non-optimized) and HMAC-SHA-256 offered less permanent storage usage in all the devices.

Despite HMAC-MD5 being the best performer regarding throughput, it is still much slower than top cipher performance in all the devices. In the Waspmote, the faster cipher (SOSEMANUK) performs about 8 times faster than HMAC-MD5 while in the DETPIC32 and Raspberry Pi the ratio is of 7,36 and 5,38 respectively. At the same time RAM usage is also higher with a ratio of 6 between the primitive that uses least RAM and HMAC-MD5 in the Waspmote.

Primitive selection

Results also allow the inference of other conclusions such as the best primitives for use in specific scenarios taking in consideration the most restricted device: the Waspmote. DETPIC32, despite being an embedded system, provides a much higher set of requirements than the Waspmote offering a ratio of 16 and 4 between volatile and permanent memory respectively and the results confirm its much higher CPU capabilities: comparing the faster cryptographic primitives in each device, the DETPIC32 offers a throughput almost 63 times superior to the Waspmote. These results, as previously stated, are obtained using the default tools and, since the optimization flags used by the Waspmote IDE are unknown, it may be unfair to do such direct comparison between throughputs however, by default, that is the ratio obtained. Likewise, the Raspberry Pi is a much more resourceful device than both Waspmote and DETPIC32 and the concept of embedded system does not even apply to this device.

In a heterogeneous environment such as IoT may be necessary for devices with higher sets of capabilities (in this case DETPIC32 and Raspberry Pi) to perform some self-sacrifices and not use the primitives where they obtain best results but the primitives where more constrained devices perform better leading to an equilibrium of the system. The system will not perform better if communication capability between regular and constrained devices increases while decreasing in the reverse direction, being preferred a balanced strategy between parts to benefit the system. For this reasons, results obtained in the Waspmote are more significant for primitive selection and best performers should be worthy of study over the best performers for the other used devices.

Both SOSEMANUK and HC-128 performed remarkably well regarding throughput in the Waspmote (in fact in all the devices). Such algorithms, however, use respectively around 36% and 57% of RAM and 35% and 19% of storage. This is an huge drawback to their applicability in generic scenarios whereas in situations where high throughput or low power consumption is required and there is plenty of both memories available there is no inconvenience in using them.

Rabbit and AES fall into a middle category where the memory/throughput trade-off is not so accentuated. Meanwhile, it should be noted that Rabbit offers around twice throughput than AES-128, but also uses more storage than both AES versions. Such difference might just be symbolic since it represents an increase in permanent memory from 1,04% in AES-256 to 2,83% in Rabbit (with AES-128 in the middle featuring 2,08%) in the Waspmote. Rabbit also offers a higher throughput in remaining devices possibly placing it ahead of an AES choice except when the little storage increase is problematic.

Lastly, there are algorithms that offer very little impact in memory while still maintaining a medium throughput: RC5 and XTEA. These are algorithms that should be used when memory footprint affects functionalities or when resources reserved for security are sparse (sadly not uncommon). While the optimized version of XTEA uses a significant amount of storage, its "regular" version uses less than 1% of the Waspmote's storage and the same is visible with RC5. RC5 has the throughput leadership when comparing both algorithms but also uses a larger percentage of the Waspmote's RAM (2,15%) than XTEA (0,95%). Results are also consistent with the ones obtained in both DETPIC32 and Raspberry Pi.

Figure 4.1 represents a full comparison between the mentioned algorithms throughputs as well as memory footprint including the already mentioned best HMAC performers in the Waspmote device.

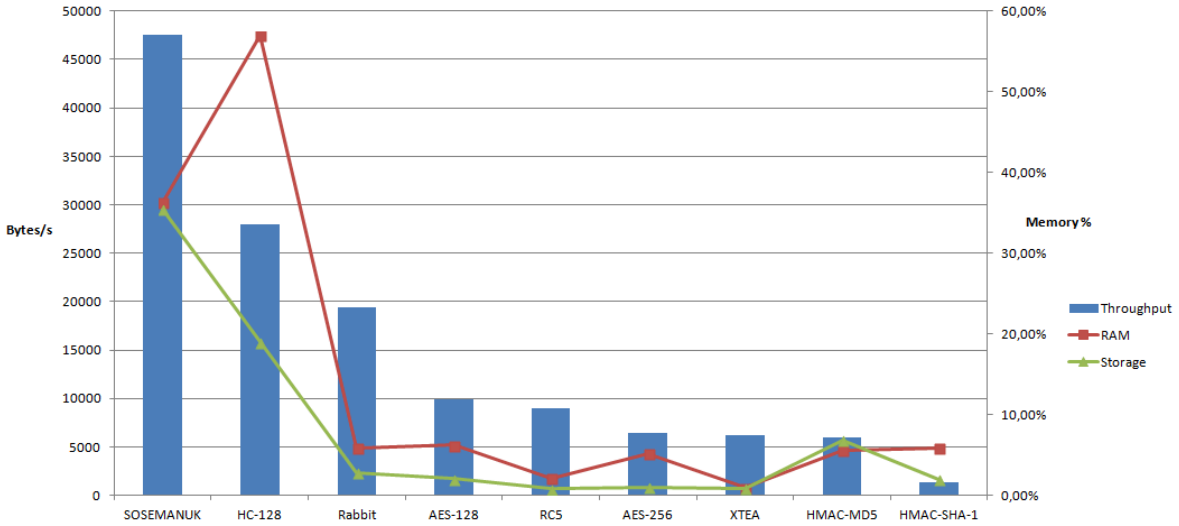


Figure 4.1: Representation of the most relevant performers in the Waspmote device.

Given the current set of tested primitives and methodology and the Waspmote device, the best performers, or in other words, the performers who are more prominent in a compromise between throughput and memory footprint or in a specific application, are hereby declared. Other primitives, including some top 5 primitives did not achieve such compromise and were not represented. Results from the majority of the selected primitives are also transposed to both the DETPIC32 and Raspberry Pi devices. SOSEMANUK and HC-128 also feature these devices top 5 regarding throughput, also being in a high throughput/high memory category. In the same way, RC5 and XTEA continue to provide a medium throughput (with more relevance on RC5 where the throughput differences are more accentuated) while still having a low memory footprint. Likewise, HMAC-MD5 is the fastest HMAC in all the devices and SHA-1 the one offering less permanent memory consumption. Both AES version results are, however, inconclusive. While performing along with RC5 and XTEA in the Waspmote regarding throughput, their performance drops on both DETPIC32 and Raspberry Pi at the same time that permanent memory footprint also increases comparatively with other primitives. The classification of such results as inconclusive is due to the large number of AES implementations available whereas some may favor architectures with 8-bit words (Waspmote) and others with 32-bit (DETPIC32 and Raspberry Pi). This is just a possibility and no further tests were performed with different AES implementations, leaving such results simply as inconclusive.

Initialization vs non-initialization in stream ciphers

Despite what it may seem, constant reinitialization of a particular cryptographic primitive may prove useful in some real world applications. Given a scenario where a device actually powers off— not entering idle mode, actually being shutdown— and then activated by an external unit (e.g. RTC) in order to transmit information, RAM memory contents would be lost. Such situation may seem abstract but could indeed happen when the communications

are significantly spaced. And, in such cases, the initialization time of primitives may play a significant role.

Such type of initializations greatly affected the results obtained with stream ciphers. Values range from losses of around 99% in performance to a mere less than 1%. Th HC family— HC-128 and HC-256— which performed distinctively in all the devices is the one suffering the most accentuated decline: the least decline is observed around 98% in the DETPIC32 with others being in orders between 98% and 99%.

SOSEMANUK, which is also included in the top 5 for all the devices, experiences losses between 64,66% in the DEPTPIC32 and around 84% in the Waspmote with Raspberry Pi standing in the middle with a decrease of throughput of 77,15%.

Rabbit and Trivium, featuring the top 5 performance in all the devices and Raspberry Pi respectively, suffer a cutback around 60% in all devices.

Salsa20/12, in contrast, suffers very little from being repeatedly initialized, with a maximum decrease of 2% in the Raspberry Pi and not being in the top 5 but in 6th place for this device. Also important to refer is that this algorithm is one of the fastest for the DETPIC32.

Remaining primitives reduce throughput between the orders of 20% and 40% not having additional patterns worth noting.

Worthy of note is the recalling that such results were obtaining for 80 byte sets of data. Variations in such amount would certainly account for different losses. Simply because the Salsa20/12 reports less performance degradation when constantly reinitialized is no reason to always use it without first assessing message sizes to be transmitted. Regardless, for contents of 80 bytes or less (where initialization time will become more accentuated) its use should be pondered (especially when considering the use of a stream cipher) since the constant reinitialization of stream ciphers alters the top 5 throughput performers.

Chapter Five

Security in a complete M2M solution

Observed and analyzed how devices behave when executing uniquely cryptographic code, it's necessary to advance further in the direction of a real scenario. A scenario which does not comprehend only code destined to enforce security but that being just part of a larger set of features. The flexibility of cryptographic primitives may play a fundamental role in the end narrowing algorithm possibilities or at least help grasping just how much of the devices' resources will be available for implementing security after the remaining (and visible for the final user) functionalities are implemented.

It's certain that sometimes industrial manufacturers implement proprietary solutions which serve their purposes and could also result, in the end, in products having the same functionalities as standardized options, being a design option/choice[1]. However, in this case, where the intent is to grasp the security margin after implementing functionalities, that would be a bad approach. A bad choice because defining message envelopes and resource exposure would only be applicable to a single case which would not be used in any real environment—manufacturers' proprietary solutions as proprietary as they may be will be used in real situations. Designing a solution from scratch would not successfully meet the intended purpose. Therefore an existent solution which may be largely used in the near future was chosen for resource exposure: CoAP.

5.1 ETSI M2M

Already mentioned in the beginning, there is the pursue for effective global standardization in IoT. Advantages of standardization are well known and numerous from interoperability to

[1]Not using recognized standards may (with a great certain) cause interoperability problems when integrating products which follow standards. Meanwhile that can also be the goal: only allow a subset of products to interoperate. Despite all the problems and difficulties it may cause, it's the manufacturer's option.

ease of deployment of solutions. From the advances in this field, one is particularly attractive due to following an open approach: the ETSI M2M initiative.

Responding to industry appeals and in consequence of a formal standardization mandate from the European Commission, the ETSI Technical Committee M2M emerged in 2009. Since then, this TC have been active, establishing a functional architecture following a horizontal approach for M2M supported by a considerable number of published technical specifications and elaborating a series of technical reports involving specific scenarios from eHealth to smart grids and others.

Such architecture, comprised of three distinct domains— device, gateway and network service domains— features a RESTful service provisioning approach at all layers (service capability layers— SCLs) with the ultimate goal of scalability. Besides, it also specifies a number of different interactions and interfaces for attaining such services regarding device's capabilities[64]:

- Communication between non-ETSI compliant devices/gateways and the network— legacy case;
- Non-ETSI compliant devices communicating with a compliant gateway or with other ETSI compliant devices— legacy case;
- The standard scenario of devices communicating with a gateway and the gateway with higher layers;
- Direct communication between a device and the network layer.

Due to inclusion of the legacy cases, current vertical approaches can be deployed in conjunction with an ETSI infrastructure as long as the gateway or other entities translate their actions. Such feature is of enormous importance since it means that existing solutions may adopt the ETSI architecture scalability without infrastructure replacement but simply by adding entities with proxy capabilities.

5.2 CoAP

As any other RESTful philosophy, ETSI M2M implies the existence of resources. While being an abstract concept, resources are no more than addressable entities/objects containing information— on ETSI M2M identified by an URI[65]. Besides the URI, everything from operations to be performed as well as eventual parameters required to perform actions and even the data types being exchanged have to be specified in order for both communicating parties to understand each other. For that purpose ETSI proposes the use of HTTP or CoAP also allowing other envelopes[66].

Whereas HTTP is widely used in nowadays Internet including to request resources either it be web pages or REST services, it is a much more complex protocol than it may appear at

first sight. Usage of such protocol has impact on both device's resources and communications which may be considered excessive if the main reason for its usage is the simplified and familiar access to resources through URIs following a RESTful approach. Meanwhile CoAP presents itself as a lightweight REST alternative to HTTP.

5.2.1 Brief Overview

CoAP or Constrained Application Protocol, is an application layer protocol (as well as HTTP), currently undergoing as standardization attempt by the IETF featuring its 18th draft version. As the draft's abstract indicates, CoAP is designed to be lighter than HTTP while still being easy to translate between the two protocols. There are, however, a number of features worth mentioning[67]:

Transport layer. In contrast to HTTP, designed to run over TCP, CoAP can perform over any transport layer be it stateful or not.

Binary headers. Headers are binary instead of text allowing to obtain a reduced header size with a minimum size of 4 bytes.

Resource discovery. Without prior knowledge of host's resources, it's possible to obtain them sending a request with the host URI followed by ".well-known/core".

Multicast. CoAP supports multicast in contrary to HTTP, allowing messages to be sent to multiple nodes, further reducing message overhead.

DTLS. DTLS is to CoAP as TLS is to HTTP and provides equivalent security guarantees but, as CoAP, also assuming an unreliable transport layer.

There are also a significant number of similarities with HTTP including the existence of GET/POST/PUT/DELETE methods, a request/response model and many others. Many more similarities and differences could be enumerated and CoAP further described but that is not the purpose. The purpose is simply to state that CoAP is a lightweight option for providing REST services which provides easy translation to HTTP while offering a smaller footprint both in communications and processing capabilities with features offering potential for large scale adoption in IoT devices[68].

5.2.2 Implementation

With the purpose of evaluating the CoAP overhead in devices, an implementation was placed in a device. It's important to enhance that in projects defining CoAP as a requirement, the amount of resources reserved for security may equal the quantity of resources available only after the CoAP and resources deployment. Defining an approximate estimate of resources used by such implementation becomes important to determine exactly when CoAP and secure messaging can coexist.

Already described in Chapter 4, a libelium Waspmote was selected as the target device. An existing CoAP library intended for Arduino— Arduino-CoAP[2]— written in C++ was modified to be compatible with the Waspmote. The choice of such library resided on the fact that it's a very minimalistic implementation, relies on underlying layers that not UDP/IP and is not heavily dependent on a particular minimalistic operating system (such as CONTIKI or TinyOS[3]). Despite presenting strict implementation ties with Arduino and its libraries, as any libraries not depending on operating systems, does not rely on high level operations provided by their existence. It's also important to mention that Arduino-CoAP implements simply a reduced, functional version of the 8th CoAP draft being possible to retrieve resources as well as observing them. Such implementation does not contain all the CoAP features and would not be included in a fully ETSI compliant device due to limited functionalities but it's a legit implementation for devices intending to be as ETSI compliant as possible within their own limitations. Also worth noting is that DTLS is not implemented. Overhead of such library establishes the minimum requirements to access data in a CoAP fashion without the use of any cryptographic security.

Such library was also modified in order to not depend on the particular transmission method underneath CoAP— being instead an object to handle communications given as an argument for the library at initialization. Due to the use of C++ and polymorphism this was achieved modifying all device dependent interactions by interactions with an object given at initialization, object which is responsible for implementing all interactions with the communication module. Such alterations are depicted in Figure 5.1.

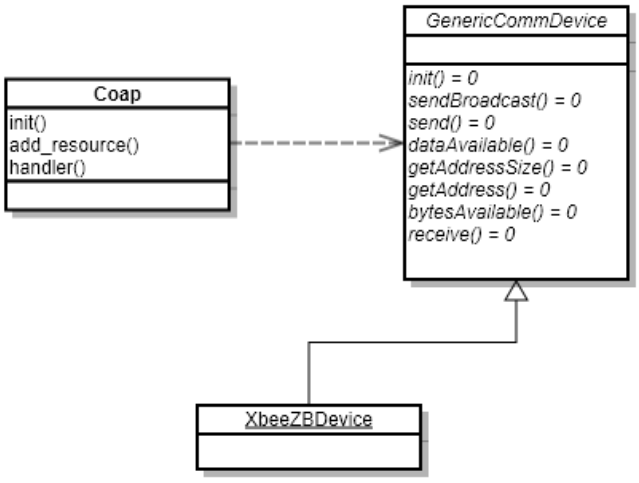


Figure 5.1: Class diagram reflecting the CoAP library changes. Methods denoted "=0" in classes represent pure virtual methods.

[2] `https://github.com/dgiannakop/Arduino-CoAP`

[3] Both CONTIKI and TinyOS are open-source implementations of minimalist operating systems intending for use in constrained devices.

Besides mediating access to communications through an abstract class, a specific scenario had to be implemented in order to test such modifications. Such choice led to the implementation of communications resorting to ZigBee. Although UDP is mentioned along the CoAP draft, as mentioned earlier, the latest draft version also clearly states when describing the CoAP message's format:

> (...)It could also be used over other transports such as SMS, TCP or SCTP, the specification of which is out of this document's scope.

Such sentence strongly discards UDP as the only underlying option and indicates that CoAP can be used on top of other transport mechanisms— and in this case, the choice laid upon ZigBee. The resulting code can, however, be used with any other communication method as long as the respective communication class is implemented extending *GenericCommDevice*.

5.2.3 CoAP overhead

Throughput was not measured during evaluation of the CoAP overhead. Each resource has a different path indicating its location (different processing time) and the obtainment of data from each resource varies in time. While the data could be obtained instantly through the use of a "dummy" resource, realistic throughput measures should include the latency in obtaining data from the resource itself— which depends on the resource (e.g. obtain values from sensors). Also, messages are obtained through a communications module which would implicate the creation of a virtual module, useless in real scenarios, to simulate message exchange. It would be possible to use a real module but the benchmark would be dependent on the conditions (including proximity). Due to this factors, throughput was not evaluated as it is considered that realistic throughput benchmarks should be performed in concrete scenarios.

Both RAM and Flash memory still consist valid measures. Meanwhile, as stated previously, memory used by resource classes as well as memory containing resource's paths varies. Determining the maximum memory usage as was determined in Chapter 4 for cryptographic primitives is very subjective due to the reason that maximum memory usage could probably be located at the class that implements the communication (maximum depth of function calls with memory being allocated by each function before calling the next level)— and that depends on the communication module. Resources and their respective classes (containing code and data) should also be considered. Besides number of resources, different resources with different management requirements can greatly influence memory usage, therefore memory measurements followed a minimum boundary approach: measuring memory required to declare and instantiate and initialize (when applicable) classes.

Besides assessing memory for an instantiation of a CoAP class, measurements were also performed resorting to its initialization with a ZigBee communication class and with CoAP resources. In order to do so, resource objects for the Waspmote's battery and accelerometer were also implemented.

It's also important to state that both RAM and flash memory usage is not linear when resorting to the use of libelium libraries. That is, when recording memory usage for two different programs, the totality of memory used by the inclusion of both functionalities in a single program may not equal the sum of both memory usages minus a base value. For instance, pieces of code used to obtain values from two different sensors may produce a different memory usage when put together even considering a base value equal to an *init* and *loop* functions stripped of functionalities. Such situations were identified during experiments using the Waspmote and are due to multiple inclusions performed by the device's libraries. In other words, if somewhere along the chain of inclusions there is a reciprocal inclusion of the same code/data definitions, such will only be included once resulting in less memory usage than the simple sum of independent pieces of code. This differs from the last chapter where cryptographic code was isolated and it was possible to retrieve a value which corresponded to the increase in memory by using cryptographic primitives. There is only an exception for this situation: an uninitialized CoAP class which does not contain resources neither a communication class different from the abstract *GenericCommDevice* (and by consequence without ties with Waspmote's libraries). These circumstances led to the decision of measuring total memory usage instead of considering an initial base referential. Memory values obtained with an uninitialized CoAP class as well as the implemented resources and different combinations using the developed communication class are presented in table 5.1.

	RAM usage (Bytes)	% RAM used	Flash usage (Bytes)	% Flash used
CoAP	1386	16,92%	2762	2,11%
XBeeZBDevice	4459	54,43%	54442	41,54%
CoAP + XBeeZBDevice	5199	63,46%	55318	42,20%
Acc Resource (Accelerometer)	1502	18,33%	6738	5,14%
Battery Resource	1410	17,21%	6822	5,20%
CoAP + XBeeZBDevice + Acc	5378	65,65%	57148	43,60%
CoAP + XBeeZBDevice + Battery	5284	64,50%	57162	43,61%
CoAP + XBeeZBDevice + Acc & Battery	5445	66,47%	58504	44,64%

Table 5.1: Memory measurements of the altered CoAP library as well as ZigBee communication and resources when deployed in the Waspmote device.

Obtained CoAP results in the particularly tested device (Waspmote) are discouraging. Not due to overhead caused by the CoAP implementation itself but due to an evident increase in memory usage when using manufacturer's libraries. And yet, those same libraries are the ones in which the manufacturer provides support being programmers accountable for the use of third-party software.

Uniquely CoAP itself, as a complete REST protocol providing services does not seem to use as much hardware resources as expected being its main requirement RAM memory. The standalone use of the modified CoAP library does not hinder the use of any of the so far studied cryptographic techniques. Notwithstanding, it's important to remember that the modified library does not implement all of CoAP features. Complete CoAP implementations (although being heavily-dependent on operating systems) such as libcoap[69] report flash requirements over 25000 bytes (over 9 times the required flash for this implementation) for a simple REST CoAP application, including transport layer but also not featuring DTLS meaning that there is no cryptographic security— and such increase in functionalities may also reflect in the required amount of RAM.

Using a "real" communication class with the modified library shows disappointing values but, indeed, there must exist underlying layers in order for devices being capable of communicating. Previously developed cryptographic code presented memory values including both total memory use and difference according to an initial base code. Since such code does not include dependencies to the Waspmote library, the effect of multiple inclusions already described is not verified and the total used memory can be estimated with a simple sum. Even not considering DTLS, some of this cryptographic code can be immediately excluded from use in conjunction with CoAP and ZigBee in this device. Values around 45% of flash memory for CoAP, ZigBee and simply two resources discourage the use of further high permanent memory usage cryptographic primitives and it happens that the fastest primitive in this device also is the only using more permanent memory— SOSEMANUK. As long as no more resources are present and total flash usage does not reach its maximum, it's still perfectly possible to use SOSEMANUK in the described situations. However, more resources would require a new assessment of the total flash memory used.

While permanent memory depletion is an easily identifiable situation (the code cannot be uploaded to the device), RAM usage is more preoccupying. RAM depletion, in the middle of a program's execution in a embedded system, is contrary to the principles of embedded systems themselves since they are developed for operation without human intervention leading to an increase in stability requirements. And, a minimum and not maximum value of almost 67% for CoAP, ZigBee and two resources is not reassuring. A percentage of 33% for RAM usage, besides not providing a memory margin for two previously analyzed primitives— SOSEMANUK and HC-128— is indeed preoccupying if the amount of resources were also to increase.

The described situation may be purely abstract— there is no guarantee that other manufacturers and devices would use such massive percentages of a device's memory simply for providing communication— but, once again, shows that real implementations are subdued to such kind of problems. Nonetheless, the minimalist altered CoAP library usage seems perfectly reasonable in other situations where underlying layers don't use high percentages of the device's memory resources.

Meanwhile Waspmote and similar devices, even using this minimalist CoAP implementation with low memory footprint underlying layers are still far from even being close to become ETSI compliant. Besides the obvious lack of CoAP features and DTLS not being implemented, ETSI M2M also defines a large set of resources that should be available in the device's SCL. Observing how this device reacts to CoAP (17% RAM usage) without any resources, adding a transport layer, DTLS, missing CoAP functionalities and even a large set of resources seems unrealistic. Still, much more work and development would be required in order to affirm this since a personal view based on experience in working with this device lacks scientific method. What preliminary obtained data shows is that simply providing a comfortable RESTful philosophy in devices uses at least 17% of volatile memory whereas the simple addition of cryptographic primitives causes an increase of less than 1% of this hardware resource for data privacy and less than 6% for integrity and authentication. Such comparison may however be unfair since the simple use of cryptographic primitives is not a security solution on its own and it would be necessary such a solution in order to compare both features at the same level.

Chapter Six

Conclusions

The main goals of this dissertation, which consisted in the study and discussion of security in IoT devices focusing on cryptographic security and evaluation of coexistence of secure messaging and resources, were fulfilled. For that, a small C cryptographic library consisting of 20 different implementation of 15 different cryptographic primitives 4 of which were block ciphers, 7 stream ciphers and 4 HMACs was created with the particularity of being portable and thus suitable for use in IoT heterogeneous scenarios. A minimalistic C++ CoAP implementation was also adapted allowing deployment in multiple scenarios with the underlying communication layers not being restricted to a particular scenario.

Focus, as it is obvious by the increase in level of detail, was in real placement of cryptographic primitives in devices that could serve the future Internet of Things or at least simulating an heterogeneous environment maintaining similar properties. From there, besides the usual AES, three not so known primitives were identified as offering more performance at similar or larger expense in both volatile and permanent memory: SOSEMANUK, HC-128 and Rabbit. Two other not so common primitives were identified offering less performance than AES but offering very attractive memory requirements: RC5 and XTEA. Their lack of wide adoption in computer environments may not reflect in embedded IoT devices due to their small footprint while also presenting good throughput rates. Other cryptographic algorithms were also presented and this work may serve as a small catalog or at least as a starting point when deciding which algorithms should be deployed in similar devices or are at least worthy further evaluations.

The conclusion that HMAC authentication offers real limitations when comparing its throughput to ciphers was also reached— in fact it was long been reached in the past— but its limitation were presented with real values in IoT devices. The existence of real values in real, heterogeneous devices offering concordant results is a confirmation that encryption and authentication should be offered by other techniques such as authenticated modes of operation when both are required. These modes and their performance were not evaluated here, consisting in a limitation of the present work.

A complete comparison among resources reserved for security and features in a complete

ETSI M2M compliant device was not possible due to the non-existence of such device and lack of implementation. In order to implement a totally compatible ETSI library for use in devices would be required another dissertation with that as its sole goal. Instead, an evaluation of CoAP hardware resources usage is presented which may have some applicability by presenting real values of the impact of creating service capability layers using such protocol. Conclusions, as expected, show that creation of such layer even without underlying layers neither resources, potentially uses much more hardware resources than the standalone use of cryptographic primitives. This comparison may be unfair since sole use of primitives is not enough to guarantee secure messaging— being required a complete protocol— but as mentioned, such comparison is performed with a stripped down (without transport layer, lacking of resources to expose and without DTLS) simple declaration of a CoAP object responsible for handling requests and responses.

6.1 Future work

This work does not, by all means, shows exactly how to obtain security in IoT devices. Security refers to much more than simply cryptography and many other directions could have been followed and explored when intending to analyze such property, from safe programming options and rules to physical protection and even security protocols assuming that the best cryptographic primitives were already established and provided no critical flaws.

Such directions constitute further possible work. Being a secure IoT environment a conjugation of several factors, all need to be studied and deployed before even considering uniting the words "security" and "IoT".

A situation was also described at the start of this dissertation that may prove countless research opportunities in the future: malware in IoT. Constrained devices are sadly unprepared to deal with malware, and IoT is a vision where those vulnerable devices will be at range of attackers and malware, through the nowadays common Internet, which is not quite secure even for mighty, powerful machines filled with anti-malware techniques. From effective malware injection (for research purposes only) to proposal of techniques for malware scanning and report, multiple exploration options are available.

As a direct consequence of the developed work and not just addressed issues, much more could also be performed including but not limited to:

- Expanding the current primitive catalog to also include comparison with asymmetric techniques and ciphers working in different modes of operations— with a special emphasis on authenticated modes;
- Integration of DTLS and the adapted CoAP library, evaluating its full hardware requirements;
- Development of an ETSI M2M device scenario, implementing resources within a CoAP library also featuring DTLS and evaluation of the hardware requirements for using such solution with and without secure messaging.

References

[1] M. Zorzi, A. Gluhak, S. Lange, and A. Bassi, "From today's INTRAnet of things to a future INTERnet of things: a wireless- and mobility-related view", *Wireless Communications, IEEE*, vol. 17, no. 6, pp. 44–51, 2010.

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: a survey", *Computer Networks: The International Journal of Computer and Telecommunications Networking*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (IoT): a vision, architectural elements, and future directions", *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[4] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder, "Management of resource constrained devices in the Internet of Things", *Communications Magazine, IEEE*, vol. 50, no. 12, pp. 144–149, Dec. 2012.

[5] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, "IOT Gateway: bridgingwireless sensor networks into Internet of Things", in *8th International Conference on Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP*, Dec. 2010, pp. 347–352.

[6] F. Yang and C. Yan, "Design of WSN gateway based on ZigBee and TD", *2010 International Conference On Electronics and Information Engineering (ICEIE)*, vol. 2, pp. 76–80, Aug. 2010.

[7] B. da Silva Campos, J. J. P. C. Rodrigues, L. M. L. Oliveira, L. D. P. Mendes, E. F. Nakamura, and C. M. S. Figueiredo, "Design and construction of a wireless sensor and actuator network gateway based on 6LoWPAN", *2011 IEEE International Conference on Computer as a Tool (EUROCON)*, pp. 1–4, Apr. 2011.

[8] H. K. Kalita and A. Kar, "Wireless sensor network security analysis", *International Journal of Next-Generation Networks*, vol. 1, no. 1, pp. 87–105, 2009.

[9] J. P. Walters, Z. Liang, W. Shi, and V. Chaudhary, "Wireless sensor network security: a survey," in book chapter of security", in *Distributed, Grid, and Pervasive Computing, Yang Xiao (Eds*, CRC Press, 2007.

[10] H. Orman, "The morris worm: a fifteen-year perspective", *IEEE Security & Privacy*, vol. 1, no. 5, pp. 35–43, 2003, ISSN: 1540-7993.

[11] The MITRE Corporation, *2011 CWE/SANS Top 25 Most Dangerous Software Errors*, 1.0.3, Sep. 2011.

[12] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos, "Memory errors: the past, the present, and the future", *15th International Symposium on Research in Attacks, Intrusions and Defenses*, Sep. 2012.

[13] McAfee, *Mcafee threats report: fourth quarter 2012*, 2012.

[14] ——, *Mcafee threats report: first quarter 2013*, 2013.

[15] Z. Jianwei, G. Liang, and D. Haixi, "Investigating China's online underground economy", University of California, Institute on Global Conflict and Cooperation, Tech. Rep., Jul. 2012.

[16] N. Falliere, L. O. Murchu, and E. Chien, "W32.stuxnet dossier", Symantec, Tech. Rep., Feb. 2011.

[17] G. H. Nibaldi, "Specification of a Trusted Computing Base (TCB)", MITRE CORP BEDFORD MA, Tech. Rep., Nov. 1979.

[18] NASA, *Nasa automated information security handbook*, 2410.9A, Jun. 1993.

[19] H. Delfs and H. Knebl, *Introduction to Cryptography: Principles and Applications*, 2nd ed., ser. Information Security and Cryptography. Springer, 2007, pp. 01-03.

[20] S. A. D'Agostino, D. Engberg, and A. Sinko, "The roles of authentication, authorization & cryptography in expanding security industry technology", Security Industry Association (SIA), Tech. Rep., Dec. 2005.

[21] A. Zúquete, *Segurança em Redes Informáticas*, 3rd ed. FCA, 2010, pp. 55-60.

[22] ——, *Segurança em Redes Informáticas*, 3rd ed. FCA, 2010, pp. 33-35.

[23] ——, *Segurança em Redes Informáticas*, 3rd ed. FCA, 2010, pp. 38-39.

[24] J. John R. Black, "Message authentication codes", pp. 19-21, PhD thesis, University of California, Davis, 2000.

[25] ——, "Message authentication codes", pp. 25-28, PhD thesis, University of California, Davis, 2000.

[26] A. Zúquete, *Segurança em Redes Informáticas*, 3rd ed. FCA, 2010, pp. 65-67.

[27] M. Bellare, R. Guérin, and P. Rogaway, "XOR MACs: new methods for message authentication using finite pseudorandom functions", in *Advances in Cryptology — CRYPT0' 95*, 1995, pp. 15–28.

[28] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication", in *CRYPTO '96 Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, 1996, pp. 1–15.

[29] S. Turner and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, Mar. 2011, pp. 3–4. [Online]. Available: `http://tools.ietf.org/rfc/rfc6151.txt`.

[30] T. Wollinger, J. Guajardo, and C. Paar, "Cryptography in embedded systems: an overview", in *Proceedings of the Embedded World 2003 Exhibition and Conference*, 2003, pp. 735–744.

[31] S. Vanstone, "Next generation security for wireless: elliptic curve cryptography", *Computers & Security*, vol. 22, no. 5, pp. 412–415, 2003, ISSN: 0167-4048.

[32] V. Gupta, S. Gupta, S. Chang, and D. Stebila, "Performance analysis of elliptic curve cryptography for SSL", in *WiSE '02 Proceedings of the 1st ACM workshop on Wireless security*, 2002, pp. 87–94.

[33] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and RSA on 8-bit CPUs", in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds., vol. 3156, 2004, pp. 119–132, ISBN: 978-3-540-22666-6.

[34] ANACOM, *Avaliação da QoS dos serviços GSM, UMTS e cobertura das redes (GSM e WCDMA), nos principais aglomerados urbanos e eixos rodoviários de Portugal Continental*, Nov. 2011, pp. 55–60.

[35] GSMA, http://www.gsma.com/aboutus/history, available on Oct. 2013.

[36] B. H. Pansambal and R. D. Kokate, "Smart data acquisition system using M2M communication", in *International Journal of Applied Engineering Research*, 2012.

[37] D. López, I. Vázquez, J. Ruiz, and D. Sainz, "Gprs-based real-time remote control of microbots with M2M capabilities", in *Fourth international workshop on wireless information systems (WIS 2005)*, 2005, pp. 42–51.

[38] L. Cristaldi, M. Faifer, F. Grande, and R. Ottoboni, "An improved M2M platform for multisensors agent application", in *Sensors for Industry Conference, 2005*, Feb. 2005, pp. 79–83.

[39] S. Pocuca and D. Giljevic, "Machine to machine (m2m) communication impacts on mobile network capacity and behaviour", in *MIPRO, 2012 Proceedings of the 35th International Convention*, May 2012, pp. 607–611.

[40] M. Toorani and A. Beheshti, "Solutions to the GSM security weaknesses", in *Next Generation Mobile Applications, Services and Technologies, 2008. NGMAST '08. The Second International Conference on*, Sep. 2008, pp. 576–581.

[41] K. Nyberg, "Cryptographic algorithms for UMTS", in *ECCOMAS 2004, Proceedings*, vol. II, 2004.

[42] U. Meyer and S. Wetzel, "A man-in-the-middle attack on UMTS", in *WiSe 2004, Proceedings of the 3rd ACM workshop on Wireless security*, 2004, pp. 90–97.

[43] Z. Ahmadian, S. Salimi, and A. Salahi, "New attacks on UMTS network access", in *Wireless Telecommunications Symposium, 2009. WTS 2009*, Sep. 2009, pp. 1–6.

[44] O. Dunkelman, N. Keller, and A. Shamir, "A practical-time related-key attack on the KASUMI cryptosystem used in GSM and 3G telephony", in *Advances in Cryptology, CRYPTO 2010*, 2010, pp. 393–410.

[45] M. Paik, "Stragglers of the herd get eaten: security concerns for GSM mobile banking applications", in *HotMobile 2010, Proceedings of the Eleventh Workshop on Mobile Computing Systems and Applications*, ADDENDUM, JANUARY 2010, 2010, pp. 54–59.

[46] Karsten Nohl, "Rooting SIM cards", in *Black Hat USA 2013*, 2013.

[47] *Waspmote Datasheet*, http://www.libelium.com/v11-files/documentation/waspmote/waspmote-datasheet_eng.pdf, version 2.3, Waspmote v1.1 Datasheet, Nov. 2012.

[48] *Waspmote Quickstart Guide*, http://www.libelium.com/uploads/2013/08/quickstart_guide.pdf, version 4.2, Aug. 2013.

[49] NIST, *FIPS PUB 197, Advanced Encryption Standard (AES)*, U.S. Department of Commerce/-National Institute of Standards and Technology, 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[50] Intel, *Intel Advanced Encryption Standard (AES) Instructions Set*, Rev 3.01, 2012. [Online]. Available: http://software.intel.com/articles/intel-advanced-encryption-standard-aes-instructions-set/.

[51] A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique cryptanalysis of the full aes", in *Proceedings of the 17th international conference on The Theory and Application of Cryptology and Information Security*, ser. ASIACRYPT'11, Springer-Verlag, 2011, pp. 344–371.

[52] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An Ultra-Lightweight Block Cipher", in *Cryptographic Hardware and Embedded Systems - CHES 2007*, ser. Lecture Notes in Computer Science, vol. 4727, Springer Berlin Heidelberg, 2007, pp. 450–466.

[53] A. Biryukov and E. Kushilevitz, "Improved cryptanalysis of RC5", in *EUROCRYPT 1998*, Springer-Verlag, 1998, pp. 85–99.

[54] J. Lu, "Related-key rectangle attack on 36 rounds of the xtea block cipher", *International Journal of Information Security*, vol. 8, no. 1, pp. 1–11, 2009.

[55] E. Lee, D. Hong, D. Chang, S. Hong, and J. Lim, "A weak key class of xtea for a related-key rectangle attack", in *Progress in Cryptology - VIETCRYPT 2006*, ser. Lecture Notes in Computer Science, vol. 4341, Springer Berlin Heidelberg, 2006, pp. 286–297.

[56] M. Robshaw, "The eSTREAM Finalists", in *New Stream Cipher Designs- The eSTREAM Finalists*, ser. Lecture Notes in Computer Science, vol. 4986, 2008, pp. 1–6.

[57] A. Klein, "The estream project", in *Stream Ciphers*, Springer London, 2013, p. 229.

[58] I. Dinur and A. Shamir, "Breaking grain-128 with dynamic cube attacks", in *Fast Software Encryption*, ser. Lecture Notes in Computer Science, vol. 6733, Springer Berlin Heidelberg, 2011, pp. 167–187.

[59] P. Stankovski, S. Ruj, M. Hell, and T. Johansson, "Improved distinguishers for HC-128", *Designs, Codes and Cryptography*, vol. 63, no. 2, pp. 225–240, 2012.

[60] Y. Lu, H. Wang, and S. Ling, "Cryptanalysis of rabbit", in *Information Security*, ser. Lecture Notes in Computer Science, vol. 5222, Springer Berlin Heidelberg, 2008, pp. 204–214.

[61] T. Xie and D. Feng, "How to find weak input differences for MD5 collision attacks.", *Cryptology ePrint Archive*, 2009.

[62] F. Mendel, T. Nad, S. Scherz, and M. Schläffer, "Differential attacks on reduced RIPEMD-160", in *Information Security*, ser. Lecture Notes in Computer Science, vol. 7483, Springer Berlin Heidelberg, 2012, pp. 23–38.

[63] M. Stevens, "New collision attacks on sha-1 based on optimal joint local-collision analysis", in *Advances in Cryptology – EUROCRYPT 2013*, ser. Lecture Notes in Computer Science, vol. 7881, Springer Berlin Heidelberg, 2013, pp. 245–261.

[64] "ETSI TS 102 690- Machine-to-Machine communications (M2M); Functional architecture", ETSI, Tech. Rep., Jun. 2013, pp. 32–34.

[65] "ETSI TS 102 92- Machine-to-Machine communications (M2M); mIa, dIa and mId interfaces", ETSI, Tech. Rep., Jun. 2013, p. 28.

[66] "ETSI TS 102 92- Machine-to-Machine communications (M2M); mIa, dIa and mId interfaces", ETSI, Tech. Rep., Jun. 2013, p. 72.

[67] Z. Shelby, K. Hartke, and C. Bormann, "Constrained Application Protocol (CoAP), draft-ietf-core-coap-18", Internet Draft, Jun. 2013. [Online]. Available: `http://tools.ietf.org/html/draft-ietf-core-coap-18`.

[68] C. Bormann, A. Castellani, and Z. Shelby, "Coap: an application protocol for billions of tiny internet nodes", *Internet Computing, IEEE*, vol. 16, no. 2, pp. 62–67, 2012, ISSN: 1089-7801.

[69] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg, "Implementation of CoAP and its application in transport logistics", in *Proceedings of the workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, 2011.

Appendix A: Base code used in devices

Waspmote

```
#include <crypto/config.h>

volatile uint32_t operations = 0;

/* Cryptographic primitive key, data and result */
/*
uint8_t key    [] = {};
uint8_t text   [] = {};
uint8_t result [] = {};
 */

/* Initialization */
void setup()
{

  USB.begin();
  USB.println("Init");

/* Cryptographic primitive setup goes here (if necessary)*/

  /* Program timer to generate an interruption each second */
  TIMSK1 |= (1 << OCIE1A); //enable comp interrupt timer1
  sei(); //enable global interrupts

  TCCR1A = 0;
  TCCR1B = 0;
  TCCR1C = 0;
  OCR1A = 7811; // Set CTC compare value to 1Hz (Freq_Clock = 8Mhz)

  //timer startup
  TCCR1B |= (1 << CS10) | (1 << CS12) | (1 << WGM12); //prescaler = 1024
}

/* Attend interrupt */
ISR(TIMER1_COMPA_vect)
{
```

```
  //disable interrupts
  cli();

  /*Prints throughput: */
  USB.println(operations, 10);

  operations = 0;

  //enable interrupts
  sei();
}

void loop(){

  /*Base RAM is measured here*/
  //USB.println(freeMemory());

  /* Call cryptographic primitive here */

  operations++;
}
```

DETPIC32

```
#include <detpic32.h>
#include "config.h"

volatile int half_second = 0;
volatile uint32_t operations = 0;

/* Cryptographic primitive key, data and result */
/*
uint8_t key    [] = {};
uint8_t text   [] = {};
uint8_t result [] = {};
 */


/* Configure timer to generate interrupts */
void config(){
  /*timer 1 config*/
  T1CONbits.TCKPS=3;
  PR1=39062;
  TMR1=0;
  T1CONbits.TON=1;


  /*timer 1 interrupts config*/
  IFS0bits.T1IF=0;
  IPC1bits.T1IP=2;
```

```
  IEC0bits.T1IE=1;
}

int main(){
/* Cryptographic primitive setup goes here (if necessary) */

  config();
  EnableInterrupts();

  while(1){
    /* Call cryptographic primitive here */
    operations++;
  }

  return 0;
}

/* Attend interrupt */
void _int_(4) isr_T1(){

  DisableInterrupts();

  if(half_second){
    printInt(operations,10);
    printStr("\n");
    half_second = 0;
    operations = 0;
  }
  else{
    half_second = 1;
  }

  IFS0bits.T1IF=0;
  EnableInterrupts();
}
```

Raspberry Pi

```
#include "config.h"
#include <signal.h>
#include <sys/time.h>
#include <stdio.h>

typedef void (*sighandler_t)(int);

volatile uint32_t operations = 0;

/* Cryptographic primitive key, data and result */
/*
 * uint8_t key[]    = {};
```

```
 * uint8_t text[]   = {};
 * uint8_t result[] = {};
 */

void timer(int sig)
{
  printf("%d operations\n", operations);
  operations = 0;
}

int main(){

  struct timeval my_value    = {1,0};
  struct timeval my_interval = {1,0};
  struct itimerval my_timer  = {my_interval,my_value};
  setitimer(ITIMER_REAL, &my_timer, 0);

  signal(SIGALRM, (sighandler_t) timer);

  /*Init cryptographic primitive if necessary*/

  while(1){
    /*Call cryptographic primitive*/
    operations++;
  }

  return 0;
}
```

Appendix B: Views on obtained results

Cryptographic primitives sorted by performance

Waspmote

HC-256 is not featured due to inability of the Waspmote of executing the implemented version (lack of RAM memory). The remaining results can, however, be observed in table 1. It's important to note that stream ciphers were initialized prior to benchmarking.

	Bytes/s	Cycles/byte	Primitive type
SOSEMANUK	47504	168	Stream cipher
HC-128	27940	286	Stream cipher
Rabbit	19416	412	Stream cipher
AES-128	9920	806	Block cipher
XTEA (optimized)	9681,6	826	Block cipher
RC5	8933,6	896	Block cipher
AES-256	6416	1247	Block cipher
XTEA	6162	1298	Block cipher
HMAC-MD5	5970	1340	HMAC
Salsa20/12	5691,2	1406	Stream cipher
Trivium	4720	1695	Stream cipher
Present	2488	3215	Block cipher
HMAC-SHA-1 (optimized)	2263	3535	HMAC
HMAC-SHA-1	1369	5844	HMAC
HMAC-RIPEMD-160	1318,7	6061	HMAC
MICKEY 2.0	740	10811	Stream cipher
HMAC-SHA-256	609	13136	HMAC
MICKEY-128 2.0	520	15385	Stream cipher
Grain-128	192	41667	Stream cipher

Table 1: Sorted throughputs obtained using the Waspmote device

DETPIC32

As with the previous sorted results, it's important to note that stream ciphers were initialized prior to benchmarking.

	Bytes/s	Cycles/byte	Primitive type
HC-128	2985040	13,4	Stream cipher
HC-256	2061840	19,4	Stream cipher
SOSEMANUK	1629304	24,55	Stream cipher
Rabbit	1486280	27	Stream cipher
Salsa20/12	1030920	38,8	Stream cipher
RC5	958072	42	Block cipher
Trivium	842560	47,5	Stream cipher
XTEA (optimized)	570404	70	Block cipher
HMAC-MD5	405764	99	HMAC
HMAC-SHA-1 (optimized)	296305	135	HMAC
XTEA	286480	140	Block cipher
HMAC-RIPEMD-160	190701	210	HMAC
AES-128	112208	356	Block cipher
AES-256	57488	696	Block cipher
HMAC-SHA-1	32050	1248	HMAC
HMAC-SHA-256	26960	1484	HMAC
MICKEY 2.0	26640	1502	Stream cipher
MICKEY-128 2.0	22960	1742	Stream cipher
Present	20816	1922	Block cipher
Grain-128	3940	10152	Stream cipher

Table 2: Sorted throughputs obtained using the DETPIC32 device

Raspberry Pi

As with the previous sorted results, it's important to note that stream ciphers were initialized prior to benchmarking.

	Bytes/s	Cycles/byte	Primitive type
HC-128	10059040	70	Stream cipher
Trivium	7811280	90	Stream cipher
Rabbit	6565360	107	Stream cipher
HC-256	6531200	107	Stream cipher
SOSEMANUK	6425280	109	Stream cipher
Salsa20/12	5308160	132	Stream cipher
RC5	2862840	245	Block cipher
XTEA	2666480	263	Block cipher
XTEA (optimized)	2164320	323	Block cipher
HMAC-MD5	1869633	374	HMAC
HMAC-SHA-1 (optimized)	1172537	597	HMAC
HMAC-RIPEMD-160	966861	724	HMAC
AES-128	422400	1657	Block cipher
AES-256	268992	2602	Block cipher
Present	187456	3734	Block cipher
MICKEY 2.0	131280	5332	Stream cipher
HMAC-SHA-1	128289	5456	HMAC
MICKEY-128 2.0	111360	6286	Stream cipher
HMAC-SHA-256	97446	7183	HMAC
Grain-128	13040	53681	Stream cipher

Table 3: Sorted throughputs obtained using the Raspberry Pi device

Results corresponding to performance, in number of operations per second, between initialized and non-initialized stream ciphers. In the first, the algorithm is initialized prior to benchmarking and then, blocks of 80 bytes are ciphered by it while in the second the primitive is initialized and used for ciphering once per each block of data during benchmark. Despite presenting the number of operations per second as an indicative measure, the decrease in performance is directly correlated with byte throughput and inversely correlated with the number of cycles per byte as it is with operations per second.

Waspmote

As with previous appendices, HC-256 is not included with Waspmote results. The remaining results for this device can be observed at table 4.

	Operations/s (initialized)	Operations/s (non-initialized)	Decrease in performance
Grain-128	2,4	1,7	29,17%
HC-128	349,25	3,6	98,97%
MICKEY 2.0	9,25	6,33	31,57%
MICKEY-128 2.0	6,5	4	38,46%
Rabbit	242,7	95	60,86%
Salsa20/12	71,14	70,93	0,3%
SOSEMANUK	593,8	94,75	84,04%
Trivium	59	22	62,72%

Table 4: Comparison of throughput values of initialized/non-initialized stream ciphers in the Waspmote device.

DETPIC32

	Operations/s (initialized)	Operations/s (non-initialized)	Decrease in performance
Grain-128	49,25	35,25	28,43%
HC-128	37313	736	98,02%
HC-256	25773	302	98,83%
MICKEY 2.0	333	238	28,53%
MICKEY-128 2.0	287	171	40,42%
Rabbit	18578,5	7239,75	61,03%
Salsa20/12	12886,5	12638	1,9%
SOSEMANUK	20366,3	7198	64,66%
Trivium	10532	4506,5	57,21%

Table 5: Comparison of throughput values of initialized/non-initialized stream ciphers in the DETPIC32 device.

Raspberry Pi

	Operations/s (initialized)	Operations/s (non-initialized)	Decrease in performance
Grain-128	163	116	28,83%
HC-128	125738	2260	98,20%
HC-256	81640	861	99,06%
MICKEY 2.0	1641	1168	28,82%
MICKEY-128 2.0	1392	841	39,58%
Rabbit	82067	31486	61,63%
Salsa20/12	66352	65012	2,02%
SOSEMANUK	80316	18351	77,15%
Trivium	97641	35146	64%

Table 6: Comparison of throughput values of initialized/non-initialized stream ciphers in the Raspberry Pi device.

Primitives sorted by RAM usage

Sorted RAM usage is presented here. Due to impossibility of RAM measuring using the DETPIC32 and RAM variations in the Raspberry Pi corresponding to variations in code size, only the Waspmote results are presented. Percentages indicate the percentage of RAM occupied by cryptographic primitives only (without base code) considering the 8 KB in the device (8192 bytes).

	RAM usage	% RAM	Primitive type
XTEA	78	0,95%	Block cipher
XTEA (optimized)	78	0,95%	Block cipher
RC5	176	2,15%	Block cipher
MICKEY 2.0	400	4,88%	Stream cipher
Trivium	401	4,895%	Stream cipher
AES-256	425	5,19%	Block cipher
MICKEY-128 2.0	458	5,59%	Stream cipher
HMAC-MD5	460	5,62%	HMAC
HMAC-SHA-1	478	5,83%	HMAC
Rabbit	482	5,88%	Stream cipher
Grain-128	486	5,93%	Stream cipher
AES-128	511	6,24%	Block cipher
Salsa20/12	546	6,67%	Stream cipher
HMAC-SHA-1 (optimized)	620	7,57%	HMAC
HMAC-RIPEMD-160	630	7,69%	HMAC
HMAC-SHA-256	878	10,72%	HMAC
Present	1095	13,37%	Block cipher
SOSEMANUK	2972	36,28%	Stream cipher
HC-128	4666	56,96%	Stream cipher
HC-256	Over 8KB	N/A	Stream cipher

Table 7: Sorted RAM usage by cryptographic algorithms in the Waspmote device.

Primitives sorted by permanent memory usage

Sorted permanent memory usage is presented here. The numbers presented do not include base code size, revealing an increase in the binary size by addition of the primitives. While for the Waspmote and the DETPIC32 a correspondence between the primitive size and the device's memory is established, for the Raspberry Pi such does not happen. This is due to the binary being stored in an SD card and thus not offering a fixed-size permanent memory like the other devices.

Waspmote

	Permanent Mem. usage	% Permanent Mem.	Primitive type
RC5	1054	0,80%	Block cipher
XTEA	1116	0,85%	Block cipher
AES-256	1360	1,04%	Block cipher
Grain-128	1414	1,08%	Stream cipher
HMAC-SHA-1	2634	2,01%	HMAC
AES-128	2722	2,08%	Block cipher
Present	3044	2,32%	Block cipher
MICKEY 2.0	3322	2,53%	Stream cipher
HMAC-SHA-256	3572	2,73%	HMAC
Rabbit	3706	2,83%	Stream cipher
MICKEY-128 2.0	3914	2,99%	Stream cipher
Salsa20/12	5416	4,13%	Stream cipher
HMAC-MD5	8936	6,82%	HMAC
XTEA (optimized)	11924	9,10%	Block cipher
Trivium	15584	11,89%	Stream cipher
HMAC-SHA-1 (optimized)	23110	17,63%	HMAC
HC-128	24892	18,99%	Stream cipher
HC-256	29008	22,13%	Stream cipher
HMAC-RIPEMD-160	30004	22,89%	HMAC
SOSEMANUK	46436	35,43%	Stream cipher

Table 8: Sorted permanent memory usage by cryptographic algorithms in the Waspmote device.

DETPIC32

	Permanent Mem. usage	% Permanent Mem.	Primitive type
XTEA	2592	0,49%	Block cipher
RC5	2784	0,53%	Block cipher
Grain-128	4008	0,76%	Stream cipher
Salsa20/12	4456	0,85%	Stream cipher
MICKEY 2.0	5436	1,04%	Stream cipher
Rabbit	7052	1,35%	Stream cipher
HMAC-SHA-1	7432	1,42%	HMAC
HMAC-SHA-256	7864	1,50%	HMAC
AES-256	8808	1,68%	Block cipher
HMAC-MD5	11160	2,13%	HMAC
XTEA (optimized)	12872	2,46%	Block cipher
Present	13972	2,66%	Block cipher
AES-128	16280	3,11%	Block cipher
HMAC-SHA-1 (optimized)	18544	3,55%	HMAC
HC-128	19744	3,77%	Stream cipher
Trivium	21204	4,04%	Stream cipher
HMAC-RIPEMD-160	22268	4,25%	HMAC
HC-256	23308	4,45%	Stream cipher
MICKEY-128 2.0	36120	6,89%	Stream cipher
SOSEMANUK	54176	10,33%	Stream cipher

Table 9: Sorted permanent memory usage by cryptographic algorithms in the DETPIC32 device.

Raspberry Pi

	Permanent Mem. usage	Primitive type
XTEA	2022	Block cipher
Grain-128	3202	Stream cipher
Salsa20/12	3305	Stream cipher
RC5	3430	Block cipher
MICKEY 2.0	3779	Stream cipher
MICKEY-128 2.0	4149	Stream cipher
Rabbit	4740	Stream cipher
HMAC-SHA-1	4905	HMAC
HMAC-SHA-256	5256	HMAC
Present	5731	Block cipher
HMAC-MD5	6019	HMAC
AES-256	6775	Block cipher
Trivium	9428	Stream cipher
HMAC-RIPEMD-160	10071	HMAC
AES-128	13271	Block cipher
XTEA (optimized)	14070	Block cipher
HMAC-SHA-1 (optimized)	14164	HMAC
HC-128	19703	Stream cipher
HC-256	24375	Stream cipher
SOSEMANUK	30233	Stream cipher

Table 10: Sorted permanent memory usage by cryptographic algorithms in the Raspberry Pi device.

Appendix C: Cryptographic complexity vs resources

Ciphers complexity

To ease comparison among primitives, the complexity granted by the benchmarked ciphers can be observed in figure 1. Both theoretical complexity and complexity in the worst case (considering the best cryptanalysis to date) is displayed. It should be noted that this comparison does not consider the applicability of the cryptanalysis in real scenarios neither side-channel attacks.

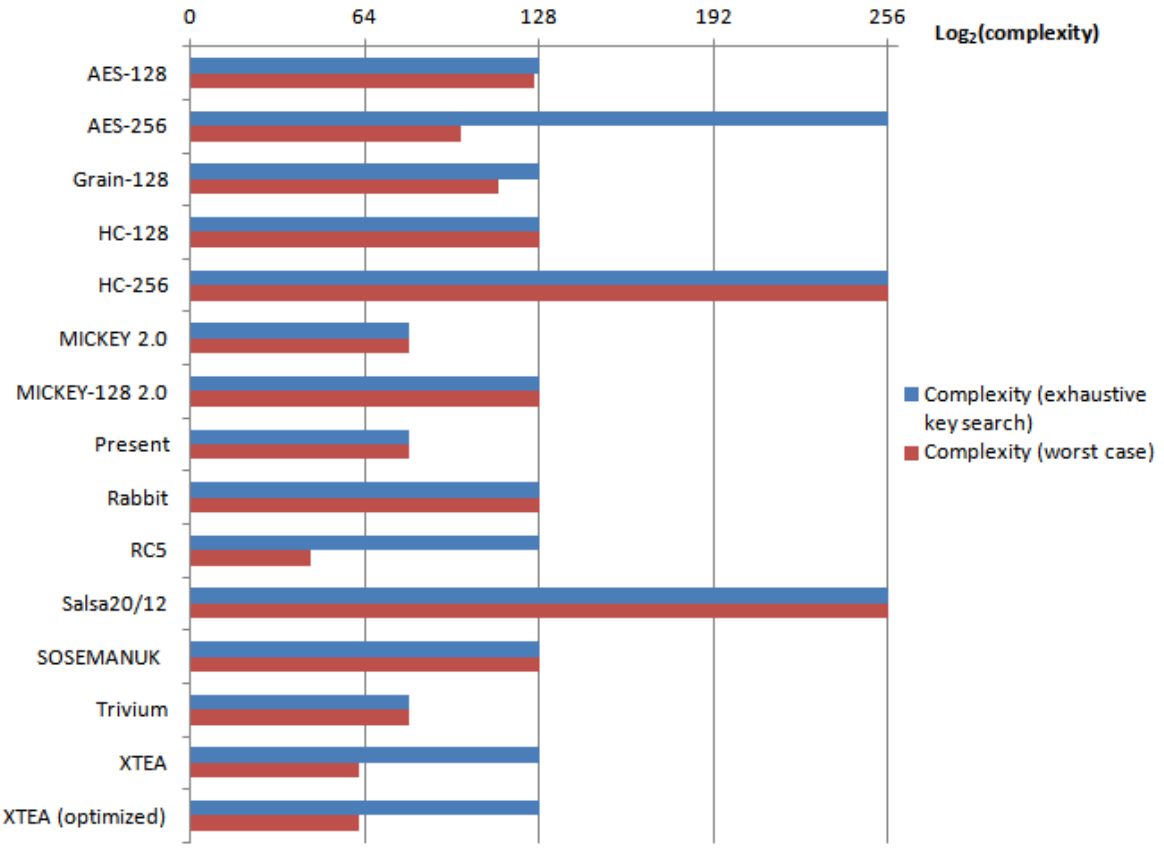


Figure 1: Complexity provided by the deployed ciphers.

Memory usage vs complexity

Figures 2 and 3 correlate both RAM and flash memory usage and complexity provided by the deployed ciphers. HC-256 RAM usage is not depicted due to the impossibility of executing it in the Waspmote device thus lacking an exact value.

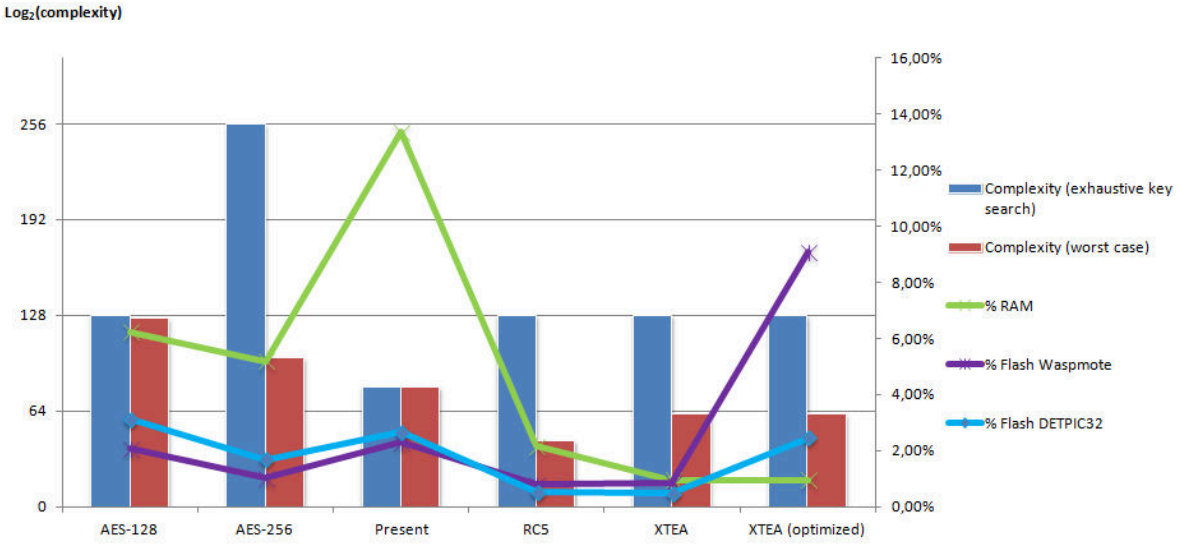


Figure 2: Correlation between benchmarked block ciphers' complexity and memory usage.

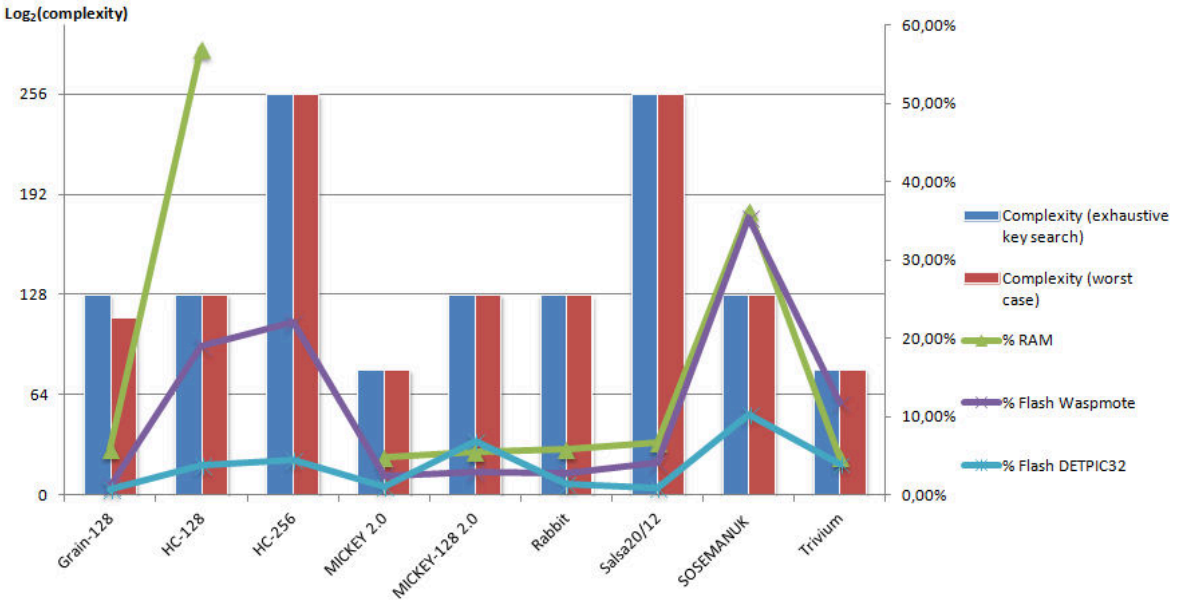


Figure 3: Correlation between benchmarked stream ciphers' complexity and memory usage.

Throughput vs complexity

Figures 4 and 5 correlate throughput and complexity provided by the deployed ciphers.

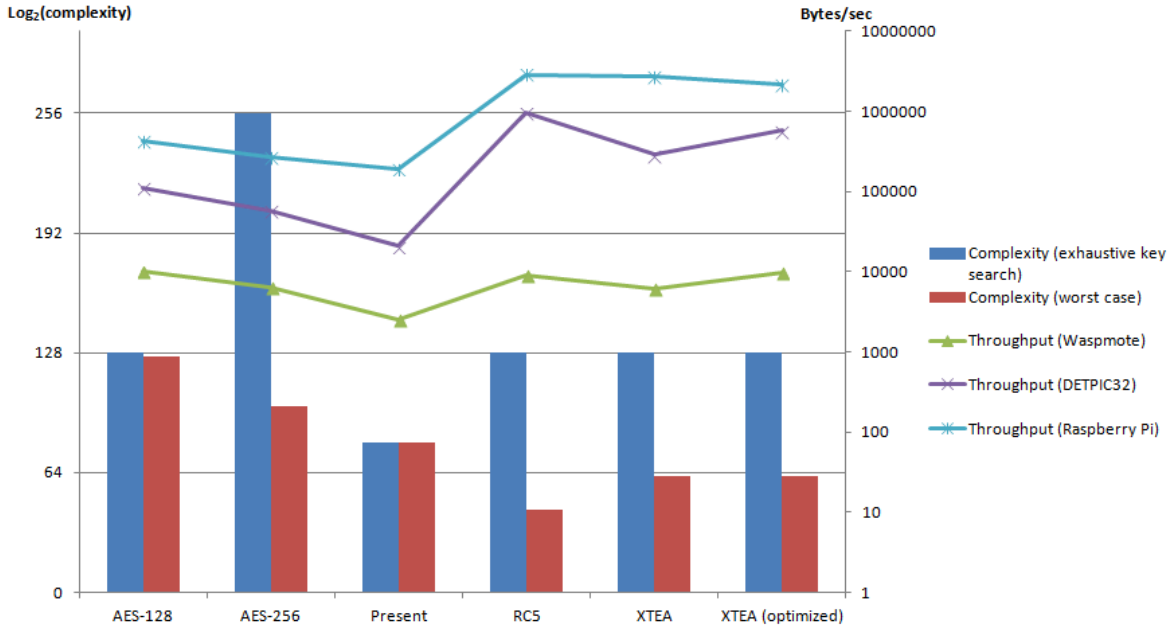


Figure 4: Correlation between benchmarked block ciphers' complexity and throughput.

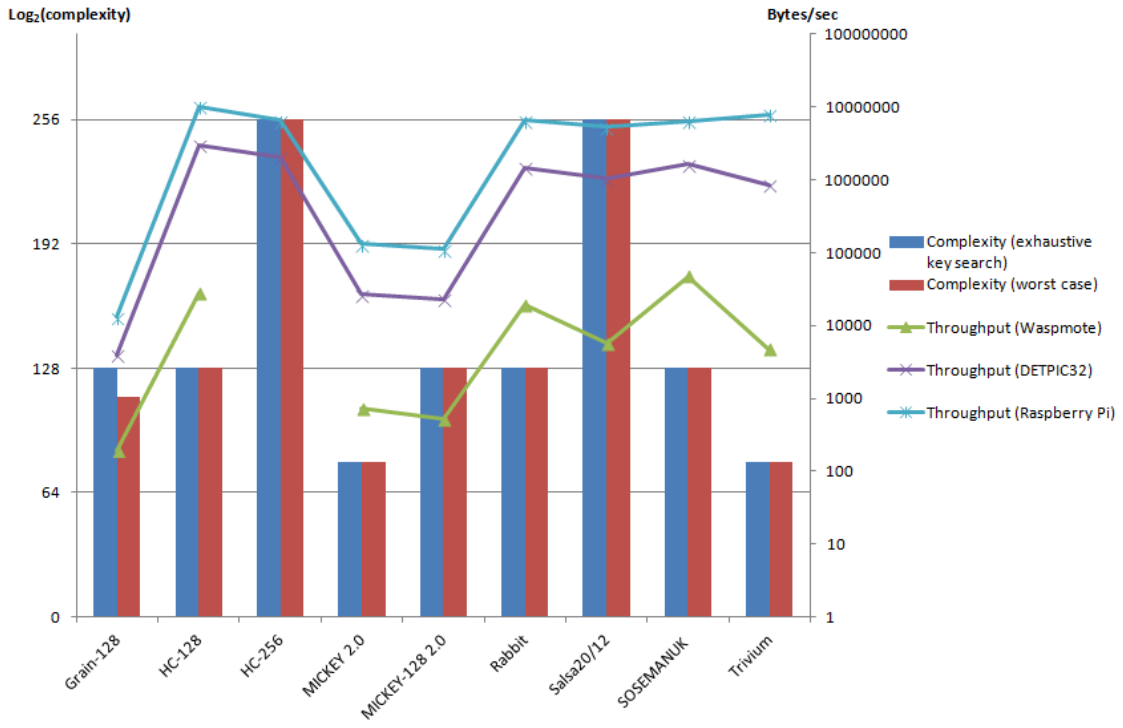


Figure 5: Correlation between benchmarked stream ciphers' complexity and throughput.